图灵程序设计丛书 

挑战程序设计竞赛（第2版）

世界顶级程序设计高手的**经验总结**

ACM-ICPC全球总冠军 巫泽俊 主译

【日】秋叶拓哉 岩田阳一 北川宜稔 著
巫泽俊 庄俊元 李津羽 译
陈越 翁恺 王灿 审



人民邮电出版社
POSTS & TELECOM PRESS



日本 ACM-ICPC参赛者 人手一册

世界顶级程序设计高手联手打造

★秋叶拓哉 昵称iwi

Google Code Jam 2010 第9名
ACM-ICPC World Finals 2012 第11名
TopCoder Open 2012 Algorithm 第4名

★巫泽俊 昵称watashi和rejudge

ACM-ICPC World Finals 2009 第6名
ACM-ICPC World Finals 2011 冠军
Google Code Jam 2012 第7名

★岩田阳一 昵称wata

Google Code Jam 2009 第3名
TopCoder Open 2010 Marathon 冠军
IPSC 2010 个人组 冠军

★庄俊元 昵称navi和navimoe

ACM-ICPC Asia Phuket Regional 2011 冠军
2012年跻身ACM-ICPC World Finals
以及百度Astar总决赛

★北川宜稔 昵称kita_masa

ACM-ICPC World Finals 2010 第16名

★李津羽

浙江大学2011级计算机系博士生
在浙大CAD&CG实验室从事科研工作



封面设计 自在书装设计 83720326@qq.com

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn


ISBN 978-7-115-32010-0
定价：79.00元



最负盛名的程序设计竞赛

★Google Code Jam（GCJ）

Google公司举办的一年一度的程序设计竞赛。

- 影响力最大、参赛面最广的程序设计竞赛。
- 每道题都有Small和Large等不同规模的测试数据。
- 自由选择喜欢的工具在本地运行程序，并提交结果。

★TopCoder

TopCoder公司举办的程序设计竞赛。既有周期举办的SRM，又有一年一度的TCO。

- 在75分钟内挑战难度和分数递增的三道题。
- 每道题的得分随用时递减。
- 独有的挑战阶段，通过寻找他人程序的漏洞赚取额外的分数。
- 系统评测的结果在最后才公布。

★ACM-ICPC

美国计算机协会（ACM）主办的面向大学生的竞赛。

- 历史最悠久、最负盛名的程序设计竞赛。
- 三名选手共用一台电脑的团体比赛。
- 5个小时内挑战10道左右的问题。

引 言 1

1

2

译 者 序

3

程序设计竞赛因其涉及的知识面广，比赛形式激烈有趣，吸引了越来越多的学生参与其中。参赛

4

者不但可以从中锻炼算法设计能力，还能够提高代码编写能力。其中的佼佼者也受到了越来越多

国际知名公司的重视和欢迎。

5

本书的几位作者是世界公认的顶尖选手，在竞赛和学术领域都取得了令人瞩目的成就。他们结合

自己的专业知识和比赛经验，将自己的心得和技巧集结成书。

6

全书将不同的算法和例题按专题编排成小节，再将不同的小节由易到难分成四章，这样即便是初

7

出茅庐的新手也不会有太大的阅读障碍。书中涵盖了在程序设计竞赛中会用到的大多数算法和技

巧，并在附录中补充了书中未介绍但也比较有用的算法。在题材的安排上，作者取舍得当，主次

分明，循序渐进，不以华而不实的奇技淫巧误导读者，又具有一定深度，相信即便是经验丰富的

8

老将同样能从书中有所斩获。本书在结合例题进行讲解时，不是简单地堆砌问题和代码，而是注

重引导读者更好地理解和运用算法来分析解决问题。对于正在学习数据结构与算法的读者而言，

9

把它作为一本练习和拓展的参考书也是很好的选择。

本书在日本广受好评，还先后在台湾地区和韩国出版。近年来程序设计竞赛在亚洲发展很快，在

10

中国大陆也出版了不少相关书籍，但鲜见高质量的佳作。所以，在读到此书时，我们非常惊喜，

迫切希望中国大陆也能引进这样的好书。2012年初，我们通过作者的推特了解到了本书第二版的

11

出版，一些前辈们踊跃翻译计算机专业书籍的经历也鼓舞了我们，让我们萌生了亲自翻译此书的

念头并联系了图灵教育。非常幸运的是，图灵教育也正考虑引进此书，于是有了今天呈现在各位

12

读者面前的简体中文版。

在翻译上，我们力求做到既尊重国内选手的习惯，又符合计算机专业的表述。在修正原书中的一

13

些笔误的同时，加入了一些译者注，以方便国内读者理解。但由于译者水平有限，不足之处在所

难免，还望读者多多包涵，并不吝提出意见和建议。

14

2 译者序

在翻译过程中，秋叶拓哉、岩田阳一和北川宜稔三位作者耐心地对我们的一些疑问和笔误给予了

一一解答和确认。浙江大学的陈越、王灿和翁恺三位老师不但将我们领进了“快乐”竞赛的大门，

还拨冗审阅了译稿并提出了宝贵的意见。网上不少同好也对本书的出版给予了关切和支持。在此

谨对他们表示感谢。

巫泽俊 庄俊元 李津羽

2013年5月6日于浙江大学

如今，形形色色的程序设计竞赛层出不穷，听说过Google Code Jam、TopCoder、ACM-ICPC的读

4

者恐怕不在少数。本书要介绍的正是这类以在规定时间内、又快又准地解决尽可能多的题目为目

标的程序设计竞赛。

5

程序设计竞赛内涵丰富，即便是经验老道的程序员，要想在比赛中取得好成绩也绝非易事。要在

程序设计竞赛中取胜，不仅需要运用灵活的想象和丰富的知识得出正确的算法，还需要一气呵成

地实现并调试通过。

6

另一方面，程序设计竞赛对新手而言亦非遥不可及。为了让更多的参赛选手体会到比赛的乐趣，

大多数比赛都会准备若干面向初学者的题目。另外，即便未能在比赛中取得好成绩，通过比赛，

7

也能够使自己的能力得到有效的锻炼。最重要的是，大家能够享受到激烈的比赛带来的乐趣。

本书的作者们参加过众多程序设计竞赛，在平时的练习和学习中，也获得了各种各样的知识与技

8

巧，本书将这些知识技巧总结成册，主要介绍算法及其在相关问题中的应用。本书依照由易及难

的顺序对问题进行讲解，章节的编排也参考了主题的难易程度及其相互的联系，内容较多的主题

则按难易程度划分为多个子主题分别介绍。各个主题由算法介绍和例题讲解穿插而成。

9

只要是具有编程基础知识的读者，均适合阅读本书。书中的源代码均用C++实现，不过只用到了

其基本功能，所以即便读者不熟悉C++也不影响阅读。

10

【关于再版】

11

令人惊喜的是，本书的第1版受到了广大读者的高度评价，在此表示感谢。特别是一些并不热衷

于程序设计竞赛的读者也购买了本书。这是因为通过本书不仅可以学到算法，更能学到其设计和

12

运用的思想。这正是本书划时代的亮点。

本书第2版追加了计算几何、搜索减枝、分治法和字符串相关算法4个主题。此外还追加了方便读者

13

加深理解的练习题，并为学有余力的读者列出了书中未涉及的拓展主题，进一步丰富了本书内容。

14

目 录 1

2

目 录

3

4

5

6

7

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

第 1 章

蓄势待发——准备篇



1.1 何谓程序设计竞赛

首先，让我们来说明一下程序设计竞赛到底是什么。

顾名思义，程序设计竞赛就是以程序设计为主题举办的竞赛。世界上有解题竞赛、性能竞赛、创

意竞赛等各种各样的程序设计竞赛。本书主要介绍解题竞赛。

解题竞赛在开始时会告知选手题目的数量，选手的目标是解决其中尽可能多的题目。程序设计竞

赛中题目的形式如下。

抽 签

你的朋友提议玩一个游戏：将写有数字的 n 个纸片放入口袋中，你可以从口袋中抽取 4 次纸

片，每次记下纸片上的数字后都将其放回口袋中。如果这 4 个数字的和是 m ，就是你赢，否

则就是你的朋友赢。你挑战了好几回，结果一次也没赢过，于是怒而撕破口袋，取出所有纸

片，检查自己是否真的有赢的可能性。请你编写一个程序，判断当纸片上所写的数字是 k_1,

$k_2, \ldots, k_n$ 时，是否存在抽取 4 次和为 m 的方案。如果存在，输出 *Yes* ；否则，输出 *No* 。

限制条件

- $1 \le n \le 50$

- $1 \le m \le 10^8$

- $1 \le k_i \le 10^8$

样例 1

输入

n = 3

m = 10

k = {1, 3, 5}

输出

Yes（例如4次抽取的结果是1、1、3、5，和就是10）

1.1 何谓程序设计竞赛 3

样例 2

1

输入

n = 3

m = 9

k = {1, 3, 5}

2

输出

No（不存在和为9的抽取方案）

3

求解这个问题，可以编写如下程序。

```
#include <cstdio>
```

4

```
const int MAX_N = 50;

int main() {

int n, m, k[MAX_N];
```

5

```
// 从标准输入读入

scanf("%d %d", &n, &m);

for (int i = 0; i < n; i++) {

scanf("%d", &k[i]);
```

6

```
}

// 是否找到和为m的组合的标记

bool f = false;
```

7

```
// 通过四重循环枚举所有方案

for (int a = 0; a < n; a++) {

for (int b = 0; b < n; b++) {

for (int c = 0; c < n; c++) {

for (int d = 0; d < n; d++) {
```

8

```
if (k[a] + k[b] + k[c] + k[d] == m) {
```

f = true;

}

}

}

}

9

}

// 输出到标准输出

if (f) puts("Yes");

else puts("No");

10

return 0;

}

11

在许多比赛中，源代码一经提交就会自动编译并运行。预先准备好的输入文件将被重定向作为程

序的标准输入。通过判断程序对应的输出是否正确，来判断解答是否正确。

当然，程序的运行是有时间限制的。在大多数比赛中，运行时间限制在若干秒。一旦程序运行的

时间超过了限制，程序就会被强行结束，当做不正确的解答处理。因此，在比赛中还必须考虑高

效的解法。

例如，本题中有$1 \le n \le 50$这个条件，像上面那样单纯的四重循环的程序，不用1秒就能得出

答案。

但是，如果变成$1 \le n \le 1000$又会怎样呢？四重循环的程序即便运行很多秒也不会结束，这将被

判为不正确。不过，这道题有更为高效的解法，即便是$1 \le n \le 1000$的情况，也能够按要求求解

（将在1.6节中再讨论）。

由此，可以说程序设计竞赛是综合了以下两个要素的复合竞赛：

- 设计高效且正确的算法

- 正确地实现

并且，为了设计算法，

- 灵活的想象力

- 算法的基础知识

也是必不可少的。

1.2 最负盛名的程序设计竞赛 5

1

1.2 最负盛名的程序设计竞赛

2

程序设计竞赛有着各种各样的形式，在此，我们来介绍其中最负盛名的几个。

3

1.2.1 世界规模的大赛——Google Code Jam（GCJ）

它是Google公司几乎每年都会举办的世界规模的程序设计竞赛，参赛者要在2~3小时内解决大约4

4

道题。一旦从在线（Online）进行的几轮预选中胜出，就能够参加现场（Onsite）总决赛。该赛

事的特点是，每道题都备有Small和Large两组输入数据。即便是难度系数较大的问题，只要输入

规模足够小，依然可以简单地求解，这一形式深受广大参赛者的喜欢。另外，GCJ并不在服务器

上自动执行程序，而是要求将源代码和本地执行的结果一同提交。

5

1.2.2 向高排名看齐！——TopCoder

TopCoder公司是一家策划并举办程序设计竞赛的公司，它举办的比赛涉及多个领域。其中之一就

6

是算法（Algorithm）比赛，该赛事大致每周都以SRM（Single Round Match）的形式举办一场，

其具有以下特点。

(1)

7

在 1 小时 15 分钟的短时间内挑战 3 道题。

(2) 提交的结果在比赛结束前是不知道的，整个过程中稍有失误，就会变成 0 分。

(3) 在编码阶段（coding phase）结束后，还有一个挑战阶段（challege phase）。该阶段可以查找别

人代码中的漏洞。如果能够提供一组输入数据，使别人的程序返回错误的结果，就能得到额

8

外的分数。

其中第3条是该赛事独一无二的特点①，也是阅读别人代码的好机会。TopCoder还有一个深受大家

喜欢的等级分系统（rating system），它会依据SRM的结果给参赛选手排名。另外，TopCoder还会

9

举办一年一度的TCO（TopCoder Open）公开赛。一旦从在线进行的几轮预选中胜出，就能够参

加在拉斯维加斯②举办的总决赛。

10

---

① 随后提到的Codeforces也参考TopCoder提供了类似但不完全一样的hack功能。——译者注

② 最初几年，TCO的决赛地点都在拉斯维加斯，不过自2011年起，每年的决赛都选择在美国不同城市举办，如好莱坞、

11

奥兰多和华盛顿。——译者注

1.2.3 历史最悠久的竞赛——ACM-ICPC

ACM-ICPC是由美国计算机协会（ACM）主办的、面向大学生的竞赛，也是历史最悠久的程序设

计竞赛。这是一个三人一队的团队比赛，选手要在5个小时内解决大约10道题。因为比赛中三名

选手共用一台电脑，题量又比其他赛事多，并且多是一些实现复杂的问题，所以团队配合显得异

常重要。想要从日本参加该项赛事，首先要参加在线进行的国内预选赛，胜出后才能参加亚洲区

域赛，取得前几名的好成绩后才能够参加世界总决赛。①

1.2.4 面向中学生的信息学奥林匹克竞赛——JOI-IOI

信息学奥林匹克竞赛是学科奥林匹克竞赛的一种，是以初中生和高中生为参赛对象的程序设计竞

赛。在日本，首先要参加日本信息学奥林匹克竞赛，取得优异成绩后，才能作为日本国家队选手

参加国际信息学奥林匹克竞赛。②其他比赛都需要尽可能快地解决尽可能多的问题，而信息学奥

林匹克竞赛只要在规定时间内求解问题即可，成绩与所用时间无关，但是它相对其他比赛而言，

求解每道题所花的时间要长得多。虽然是面向中学生的比赛，每年所出问题的难度却是非常高的。

1.2.5 通过网络自动评测——Online Judge（OJ）

在互联网上，有一些被称为Online Judge的系统，它们能够自动评测以往程序设计竞赛中的题目。

利用该系统就可以练习了。另外，其中一些Online Judge也会定期举办自己的比赛，不妨去参加

一下。在此列举几个有名的Online Judge。

□ PKU Online Judge （POJ）——http://poj.org/

题库中有大量的题目。

□ 会津大学Online Judge（AOJ）——http://judge.u-aizu.ac.jp/onlinejudge/

还包含日语题。

□ Sphere Online Judge（SPOJ）——http://www.spoj.pl/

允许使用各种各样的编程语言。

□ SGU Online Contester——http://acm.sgu.ru/

具有模拟参加历史比赛的虚拟赛功能。

□ UVa Online Judge——http://uva.onlinejudge.org/

老字号Online Judge，经常举办比赛。

□ Codecorces——http://codeforces.com/

与TopCoder一样定期举办比赛，又同其他网站一样不断维护历届题库。

---

① 中国大陆的大学生若想晋级世界总决赛，通常也需要参加大陆任意赛区的网络预赛和现场区域赛并获得前几名。当然

根据规则也有可能从亚洲其他地区获得出线权，其具体规则比较复杂并可能不断变化，大家可以从网上获得最新的规

则。——译者注

② 中国大陆的中学生首先要闯过全国联赛（NOIP）、全国竞赛（NOI）和国家队选拔赛（CTSC）三关，才能参加国际信

息学奥林匹克竞赛（IOI）。——译者注

1

1.3 本书的使用方法

2

在此，就本书所涉及的内容、使用方法及注意点做一下说明。

3

1.3.1 本书所涉及的内容

本书主要讲解程序设计竞赛中的经典问题和基础算法，并介绍便捷的实用技巧。如果仅仅是死记

4

经典问题和基础算法，遇到难解的应用问题或是需要灵活想象力的问题时，仍然会难以下手。因

此，为了加深理解，我们通过选自POJ的经典题和部分原创题来介绍实践中的例子。

另外，每章末尾都备有挑战GCJ中实战题目的小栏目，里面都是精选出来的题目。尽管要找到正

5

确的解法恐怕不太容易，还是建议读者先自己试着多思考一下。在此基础上再阅读题解，能够得

到更深刻的理解。

当然，在本书所介绍的解法之外，还会有更简洁或更高效的解法。大家不妨多试着去思考一下别

6

的解法。

1.3.2 所用的编程语言

7

比赛中可用的编程语言各色各异，而C++在几乎所有比赛中都可用。它的运行速度快，库函数丰

富，因而人气很高。本书选择C++作为所用的编程语言，并基本按照g++的规范来编写源代码。

8

1.3.3 题目描述的处理

在世界规模的大赛中，理所当然是用英语来描述题目的。不过，因为题目描述中的英语不那么难，

所用的单词往往也非常有限，所以很快就能习惯。当然，这不是英语考试，字典也是允许自由使

9

用的。另外，其中有些比赛会针对日本选手提供日语版的题目描述。英语的阅读理解不是题目的

关键，因此本书的题目都与最开始的例子一样用中文概述①。

1.3.4 程序结构

10

在许多比赛中，程序都从标准输入按指定格式读入数据。输入并非问题的关键，所以本书的程序

———————————

11

① 原书为用日语描述。——译者注

都假设输入数据已经由main函数读入并保存在全局变量中，再通过调用solve函数来求解。例如

对于最初的例子，程序将变成这样。

```
// 读入输入数据后保存在这里

int n, m, k[MAX_N];

void solve() {

bool f = false;

for (int a = 0; a < n; a++) {

for (int b = 0; b < n; b++) {

for (int c = 0; c < n; c++) {

for (int d = 0; d < n; d++) {

if (k[a] + k[b] + k[c] + k[d] == m) {

f = true;

}
```

```
}
}
}
}
if (f) puts("Yes");
else puts("No");
}
```

1.3.5 练习题

每章末尾都会介绍与本章所涉及主题相关的题目。请利用它们来加深理解、巩固知识、培养实践

能力。各个主题下的题目大致是按照难易程度排列的，其中亦包含非常难的应用问题。

1.3.6 读透本书后更上一层楼的练习方法

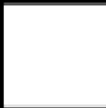
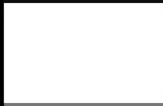
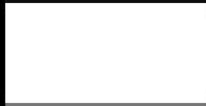
独自练习提高时，不妨同时使用Online Judge和TopCoder的Practice Room。特别是TopCoder，既

提供了解题教程，又可以阅读别人的代码，当你无论如何都想不到解法时，还可以把它们作为参

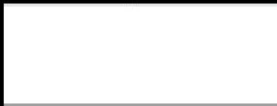
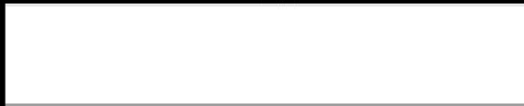
考，因而在此推荐。

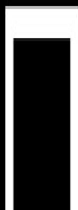
给出了各种长度的棍子

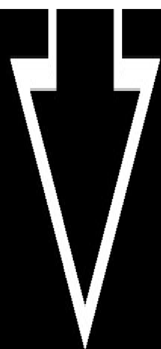
2 3 4

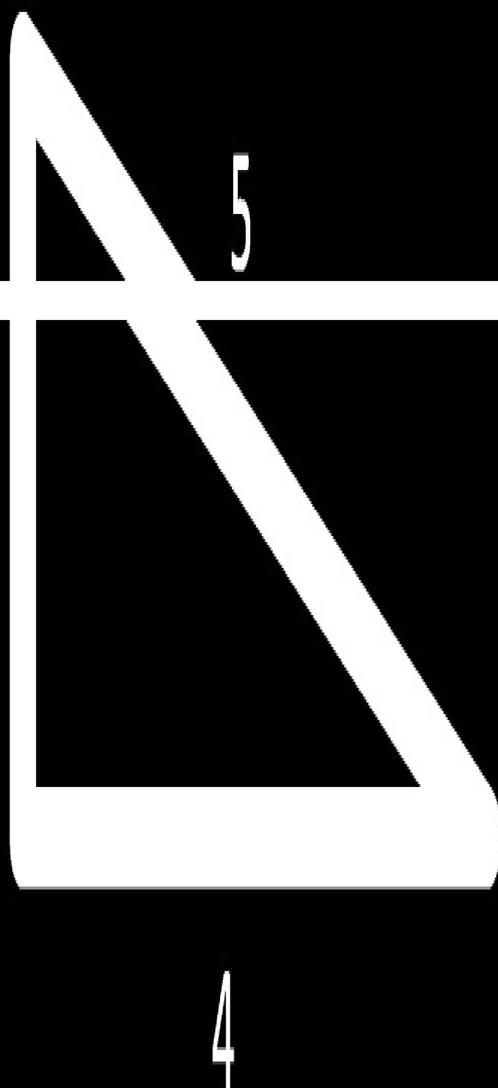
5 10





选择3根，组成周长尽可能长的三角形



3 + 4 + 5 = 12



1.6 轻松热身

本节通过对几个问题的解析，和大家一起了解程序设计竞赛中题目的风格，学习设计算法并估

算复杂度的过程。其中有一些问题比较复杂，不能想到它们的解法也不要紧，能够通过阅读题

解体会到其中的趣味就好。

1.6.1 先从简单题开始

三角形

有 n 根棍子，棍子 i 的长度为 a_i 。想要从中选出 3 根棍子组成周长尽可能长的三角形。请输

出最大的周长，若无法组成三角形则输出 0。

用5根棍子组成三角形的例子

限制条件

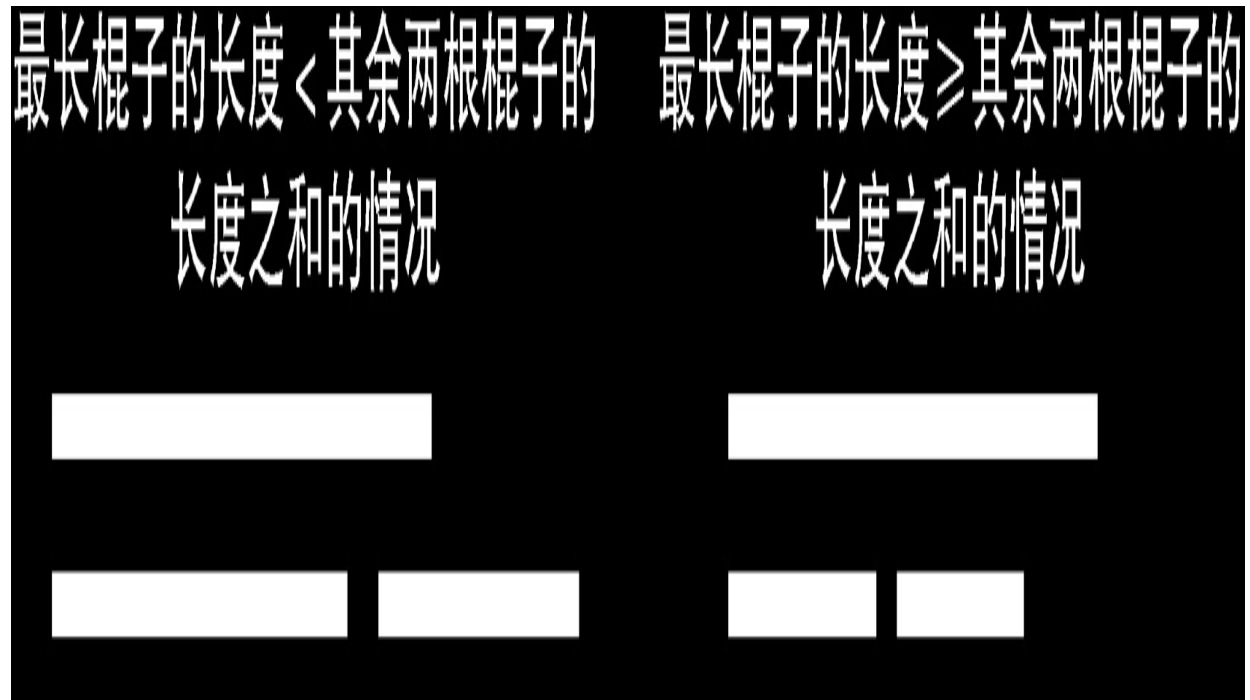
- $3 \le n \le 100$
- $1 \le a_i \le 106$

样例 1

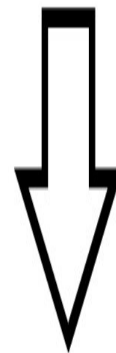
输入

n = 5

a = {2, 3, 4, 5, 10}

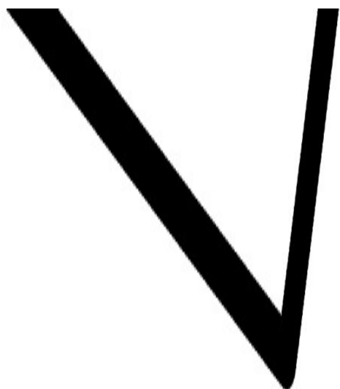














能够组成三角形　　　　不能组成三角形

输出

1

12（选择3、4、5时）

样例 2

2

输入

n = 4

a = {4, 5, 10, 20}

3

输出

0（无论怎么选都无法组成三角形）

选择3根棍子，它们能组成三角形的充要条件为

4

最长棍子的长度 < 其余两根棍子的长度之和。

5

6

7

能够组成三角形的条件

于是我们可以试想这样一种算法：首先用三重循环枚举所有的棍子选择方案，再利用上式判断能

8

否组成三角形。如果可以，那么该三角形的周长就是备选答案。

这里用了三种循环，所以复杂度是 *O* (*n* 3)。将 *n* =100代入 *n* 3得到106，可知这个复杂度是足够低

的①。

9

```
// 输入

int n, a[MAX_N];

void solve() {
```

10

```
int ans = 0; // 答案

// 让i < j < k，这样棍子就不会被重复选中了
```

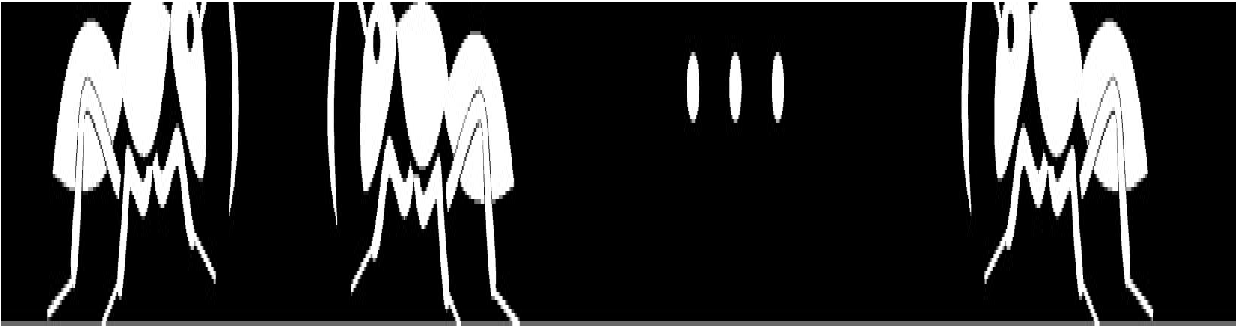
```
for (int i = 0; i < n; i++) {
```

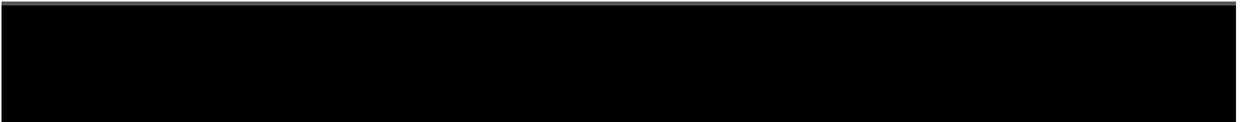
11

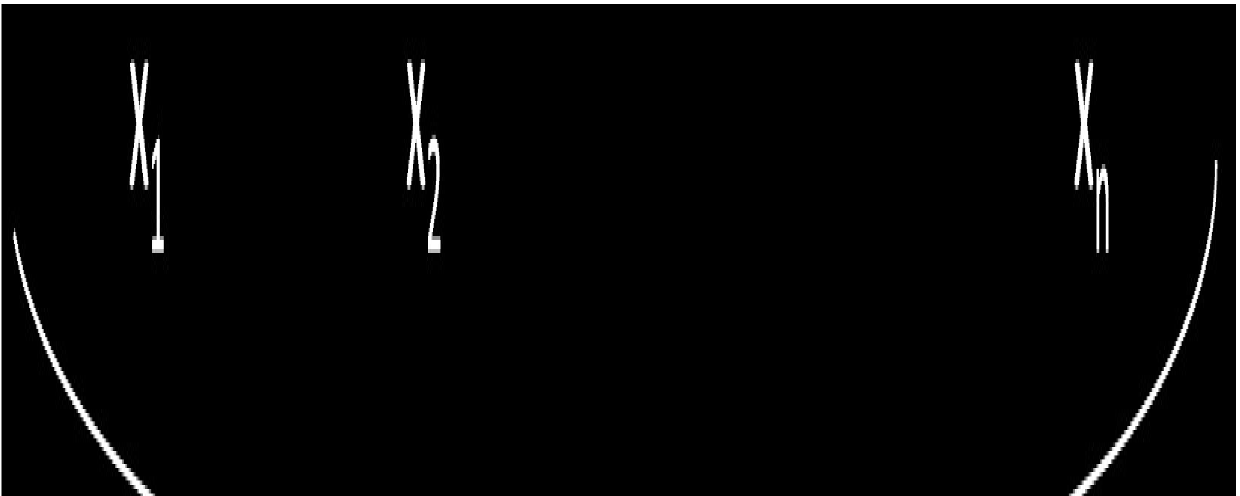
① 本题还有 $O(n\log n)$时间更高效的算法，留给有兴趣的读者思考。

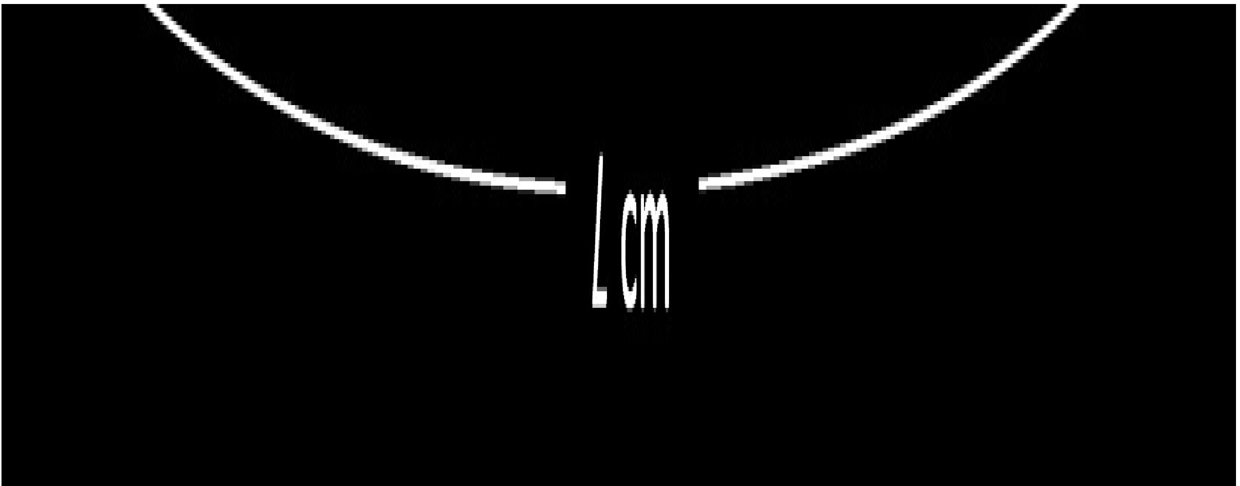
各个蚂蚁正朝向哪边是不知道的

蚂蚁 1　蚂蚁 2　蚂蚁 *n*






x_1
x_2
x_n


L cm




```
for (int j = i + 1; j < n; j++) {
for (int k = j + 1; k < n; k++) {
int len = a[i] + a[j] + a[k]; // 周长
int ma = max(a[i], max(a[j], a[k])); // 最长棍子的长度
int rest = len - ma; // 其余两根棍子的长度之和
if (ma < rest) {
// 可以组成三角形，如果可以更新答案则更新
ans = max(ans, len);
}
}
}
}
// 输出
printf("%d\n", ans);
}
```

1.6.2 POJ的题目Ants

Ants（POJ No.1852）

n 只蚂蚁以每秒 1cm 的速度在长为 L cm 的竿子上爬行。当蚂蚁爬到竿子的端点时就会掉落。

由于竿子太细，两只蚂蚁相遇时，它们不能交错通过，只能各自反向爬回去。对于每只蚂蚁，

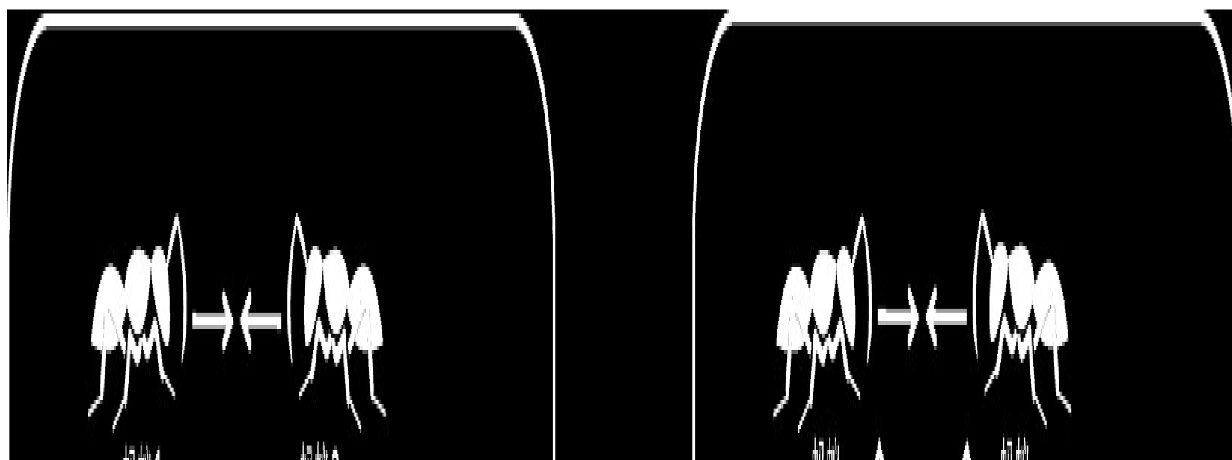
我们知道它距离竿子左端的距离 xi ，但不知道它当前的朝向。请计算所有蚂蚁落下竿子所需

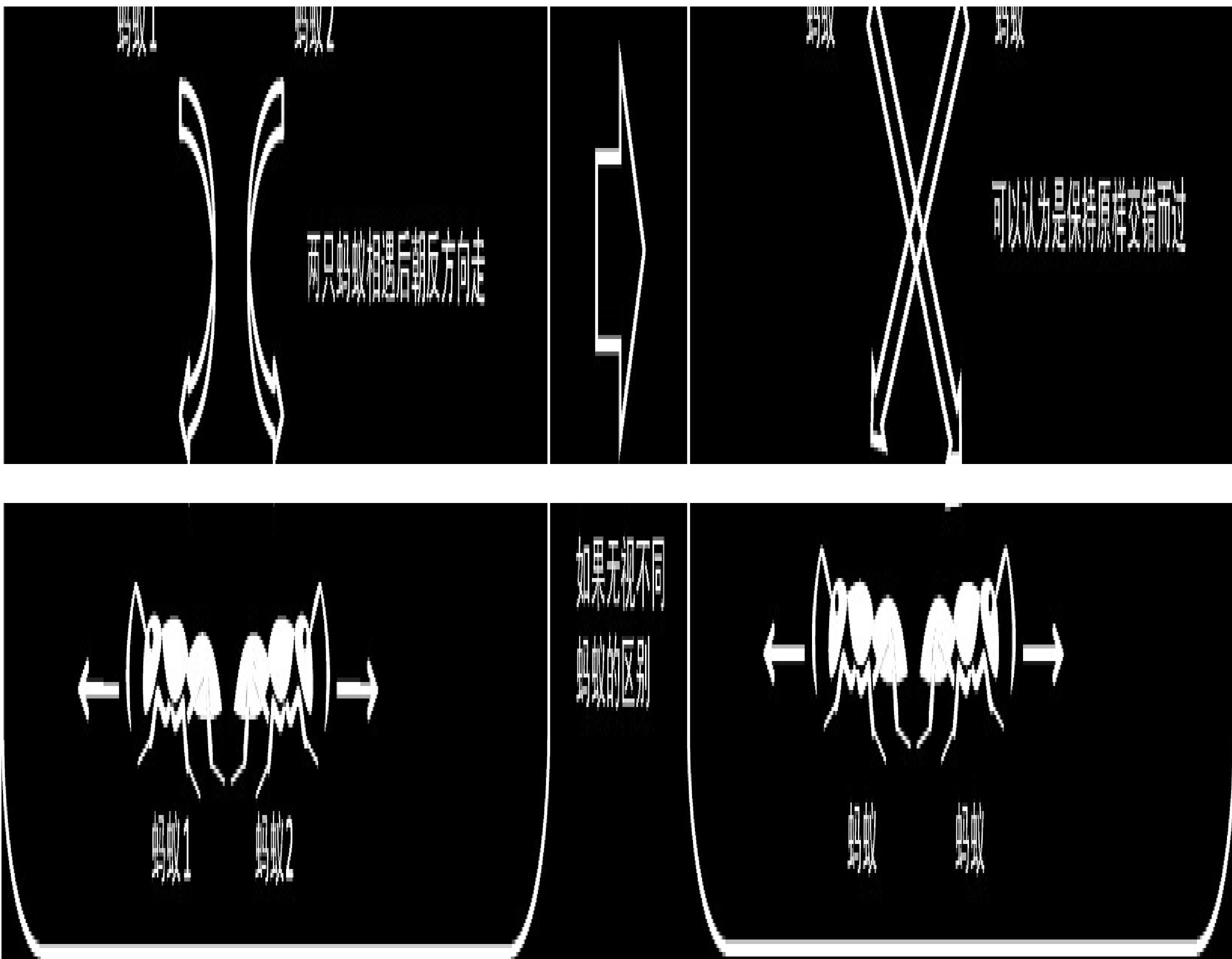
的最短时间和最长时间。

竿子和蚂蚁的情况

限制条件

- $1 \leq L \leq 106$
- $1 \leq n \leq 106$
- $0 \leq xi \leq L$





样例

1

输入

L = 10

n = 3

x = {2, 6, 7}

2

输出

min = 4（左、右、右）

max = 8（右、右、右）

3

首先很容易想到一个穷竭搜索①算法，即枚举所有蚂蚁的初始朝向的组合，这可以利用递归函数

实现（详见2.1节）。

4

每只蚂蚁的初始朝向都有2种可能， n 只蚂蚁就是2×2×…×2=2 n 种。如果 n 比较小，这个算法还

是可行的，但指数函数随着 n 的增长会急剧增长。

5

2 n 增长的趋势

n

1

5

10 20

30

100 10000

1000000

2 n 2 32 1024 1048576

109 1030 103010 10301030

6

穷竭搜索的运行时间也随之急剧增长。一般把指数阶的运行时间叫做指数时间。指数时间的算法

无法处理稍大规模的输入。

接下来，让我们来考虑比穷竭搜索更高效的算法。首先对于最短时间，看起来所有蚂蚁都朝向较

7

近的端点走会比较好。事实上，这种情况下不会发生两只蚂蚁相遇的情况，而且也不可能在比此

更短的时间内走到竿子的端点。

接下来，为了思考最长时间的情况，让我们看看蚂蚁相遇时会发生什么。

8

9

相遇后会发生什么

10

事实上，可以知道两只蚂蚁相遇后，当它们保持原样交错而过继续前进也不会有任何问题。这样

11

① 也叫蛮力搜索，口语中常简称暴搜。——译者注

看来，可以认为每只蚂蚁都是独立运动的，所以要求最长时间，只要求蚂蚁到竿子端点的最大距

离就好了。

这样，不论最长时间还是最短时间，都只要对每只蚂蚁检查一次就好了，这是 $O(n)$时间的算法。

对于限制条件 $n \leq 106$，这个算法是够用的，于是问题得解。

```
// 输入

int L, n;

int x[MAX_N];

void solve() {

// 计算最短时间

int minT = 0;

for (int i = 0; i < n; i++) {

minT = max(minT, min(x[i], L - x[i]));

}

// 计算最长时间

int maxT = 0;

for (int i = 0; i < n; i++) {

maxT = max(maxT, max(x[i], L - x[i]));

}

printf("%d %d\n", minT, maxT);
```

```
}
```

这个问题可以说是考察想象力类型问题的经典例子。有很多这样的问题，虽然开始不太明白，但

想通之后，最后的程序却是出乎意料地简单。

1.6.3 难度增加的抽签问题

如果将最开始的抽签问题中关于 *n* 的限制条件改为1 ≤ *n* ≤ 1000（题目描述参见1.1节），那么应

该如何求解呢？最初的四重循环算法是 *O* (*n* 4)时间的，将 *n* =1000带入 *n* 4得到1012，这是远远不够的，

必须改进算法。

```
for (int a = 0; a < n; a++) {

for (int b = 0; b < n; b++) {

for (int c = 0; c < n; c++) {

for (int d = 0; d < n; d++) {

if (k[a] + k[b] + k[c] + k[d] == m) {

f = true;

}

}

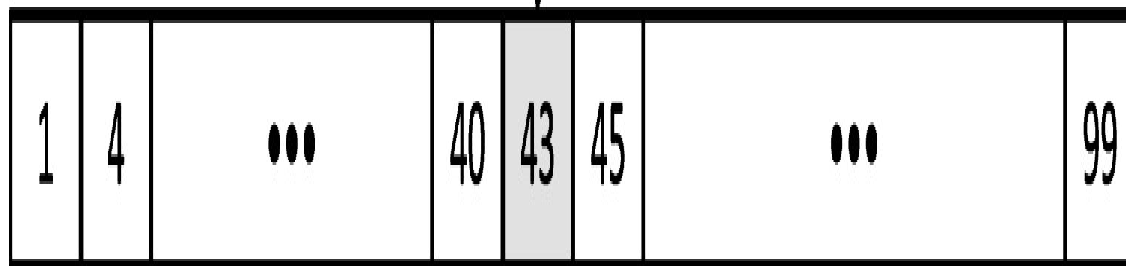
}

}

}
```

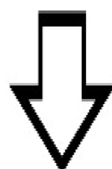
中央





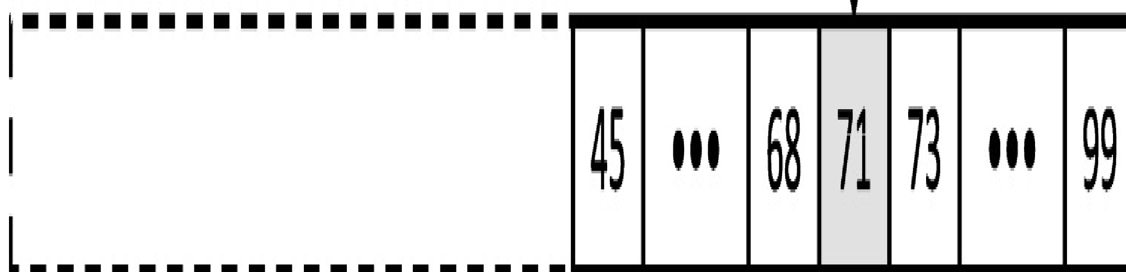


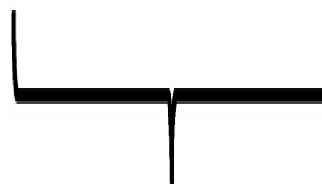
因为43<55，所以不可能在这一段



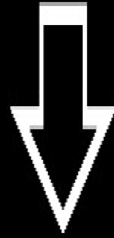
剩余部分的中央







因为55<71，所以不可能在这一段



45	•••	68



上面是最初所记载的程序的循环部分。最内侧关于 *d* 的循环所做的事就是

1

检查是否有 *d* 使得 *k* □ *k* □ *k* □ *k* □ *m a*

b

c

d

通过对式子移向，就能得到另一种表达方式

2

检查是否有 *d* 使得 *k* □ *m* □ *k* □ *k* □ *k d*

a

b

c

就是说，检查数组 k 中所有元素，判断是否有 m □ ka □ kb □ kc 。

3

让我们着眼这一点来考虑快速的检查方法。虽然也有利用数据结构优化的方法（在2.4节介绍），

这里要介绍的是名为二分搜索的算法。

1. 二分搜索与 $O(n^3\log n)$的算法

4

记所要查找的值 m □ ka □ kb □ kc 为 x 。预先把数组 k 排好序，然后看 k 中央的数字①，可知

□ 如果它比 x 小， x 只可能在它的后面半段。

□ 如果它比 x 大， x 只可能在它的前面半段。

5

如果再将上述方法运用在已经减半的 x 的存在区间上， x 的存在区间就变成了初始的1/4。这样反复

操作就可以不断缩小 x 的存在区间，最终可以确定 x 存在与否。

6

7

8

9

从数列中查找55的例子

二分搜索算法每次将候选区间减小至大约原来的一半。因此，要判断长度为 n 的有序数组 k 中是否

10

包含 x ，只要反复执行约$\log_2 n$ 次就完成了。二分查找的复杂度是 $O(\log n)$时间的，我们称这种阶的

运行时间为对数时间。即便 n 变得很大时，对数时间的算法依然非常快速。

11

① 如果元素个数是偶数，是没有中间数字的，这时候观察离中间最近的某边的值。

即使 n 变得很大，$\log_2 n$ 也依然很小

n	1	10	100	1000	10^6	10^9
$\log_2 n$	0	3	7	10	20	30

将最内侧的循环替换成二分搜索算法之后，就变成

 排序 $O(n\log n)$时间

 循环 $O(n 3\log n)$时间

$n 3\log n$ 比 $n\log n$ 大，所以这里合起来当作 $O(n 3\log n)$时间。于是，我们得到了在 $O(n 3\log n)$时间内解决

抽签问题的算法。

```
// 输入

int n, m, k[MAX_N];

bool binary_search(int x) {

// x的存在范围是k[l], k[l+1], …, k[r-1].

int l = 0, r = n;

// 反复操作直到存在范围为空

while (r - l >= 1) {

int i = (l + r) / 2;

if (k[i] == x) return true; // 找到x

else if (k[i] < x) l = i + 1;

else r = i;

}

// 没找到x

return false;

}

void solve() {
```

```
// 为了执行二分查找需要先排序
sort(k, k + n);
bool f = false;
for (int a = 0; a < n; a++) {
for (int b = 0; b < n; b++) {
for (int c = 0; c < n; c++) {
// 将最内侧的循环替换成二分查找
if (binary_search(m - k[a] - k[b] - k[c])) {
f = true;
}
}
}
}
if (f) puts("Yes");
else puts("No");
}
```


事实上，像binary_search这样的函数，多数情况下无需自己实现，可以使用标准实现。例如

1

C++的STL中就含有提供基本同样功能的函数。

2. *O* (*n* 2log *n*)的算法

2

但是，将 *n* =1000带入 *n* 3log *n* ，会发现这依然是远远不够的，必须要对算法做进一步优化。刚才我

们只着眼于四重循环程序中最内层的循环。接下来，让我们着眼于内侧的两个循环。

同刚才一样的思路，内侧的两个循环是在

3

检查是否有 *c* 和 *d* 使得 *k* □ *k* □ *m* □ *k* □ *k* 。

c

d

a

b

这种情况并不能直接使用二分搜索。但是，如果预先枚举出 *kc* + *kd* 所得的 *n* 2个数字并排好序，便可

以利用二分搜索了①。

4

该算法

□ 排序 *O* (*n* 2log *n*)时间

5

□ 循环 *O* (*n* 2log *n*)时间

总的也是 $O(n^2\log n)$时间。这样就可以确信即便 n =1000也能妥善应对了。

// 输入

6

int n, m, k[MAX_N];

// 保存2个数的和的数列

int kk[MAX_N * MAX_N];

7

bool binary_search(int x) {

// x的存在范围是kk[l], kk[l+1], …, kk[r-1].

int l = 0, r = n * n;

// 反复操作直到存在范围为空

8

while (r - l >= 1) {

int i = (l + r) / 2;

if (kk[i] == x) return true; // 找到x

else if (kk[i] < x) l = i + 1;

else r = i;

}

9

// 没找到x

```
return false;

}
```

10

```
void solve() {

// 枚举k[c]+k[d]的和

for (int c = 0; c < n; c++) {
```

____

11

① 确切地说，去除重复后 $n(n+1)/2$ 个数字就够了，不过为了方便可以写成枚举 n^2 个数字。


```
for (int d = 0; d < n; d++) {

kk[c * n + d] = k[c] + k[d];

}

}

// 排序以便进行二分搜索

sort(kk, kk + n * n);

bool f = false;

for (int a = 0; a < n; a++) {

for (int b = 0; b < n; b++) {

// 将内侧的两个循环替换成二分搜索
```

```
if (binary_search(m - k[a] - k[b])) {

f = true;

}

}

}

if (f) puts("Yes");

else puts("No");

}
```

本问题既需要二分搜索这一基础算法知识，也需要将四个数分成两两考虑的想象力。此外，像这

样从复杂度较高的算法出发，不断降低复杂度直到满足问题要求的过程，也是设计算法时常会经

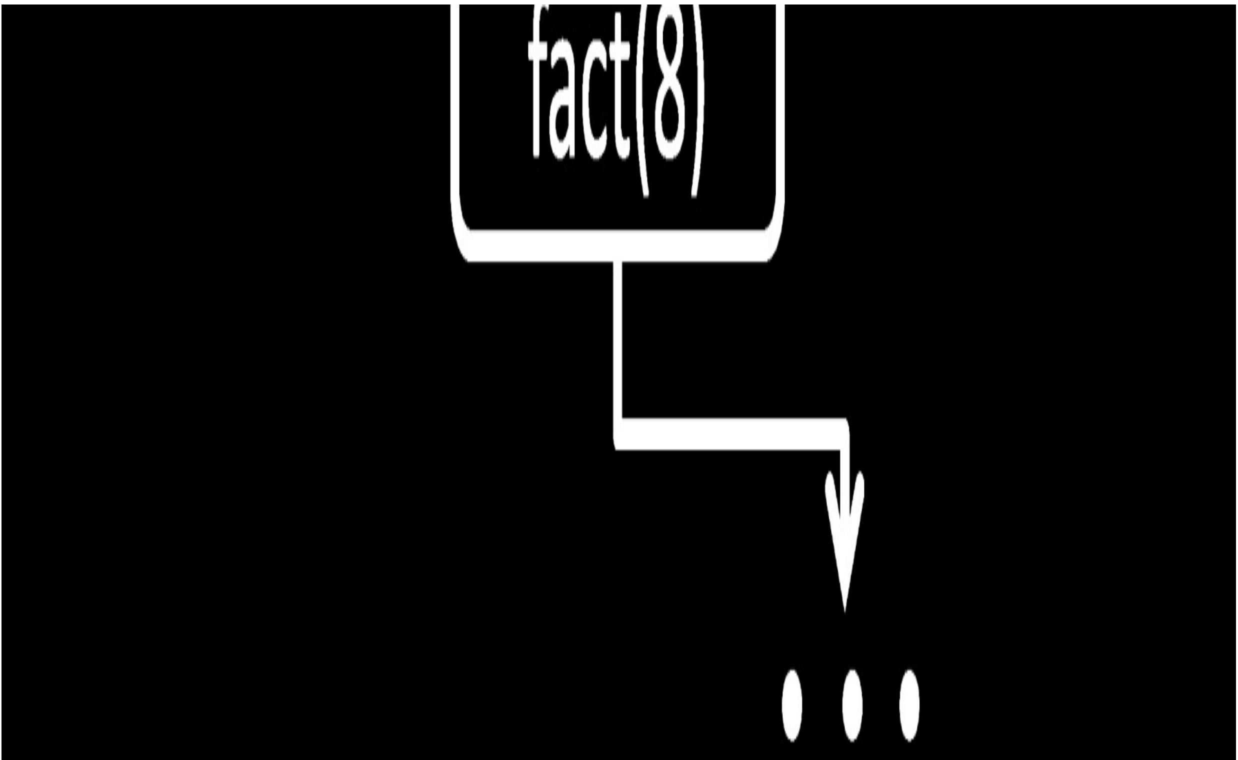
历的一个过程。

第 2 章

初出茅庐——初级篇

fact(10)

fact(9)





2.1 最基础的“穷竭搜索”

穷竭搜索是将所有的可能性罗列出来，在其中寻找答案的方法。这里我们主要介绍深度优先搜

索和广度优先搜索这两种方法。

2.1.1 递归函数

在一个函数中再次调用该函数自身的行为叫做递归，这样的函数被称作递归函数。例如，我们想

要编写一个计算阶乘的函数int fact(int n)，当然，用循环来实现也是可以的。但是根据阶

乘的递推式 $n! = n \times (n-1)!$，我们可以写成如下形式：

```
int fact(int n) {

if (n == 0) return 1;

return n * fact(n - 1);

}
```

在编写一个递归函数时，函数的停止条件是必须存在的。在刚刚的例子中，当 $n=0$时fact并不是

继续调用自身，而是直接返回1。如果没有这一条件存在，函数就会无限地递归下去，程序就会

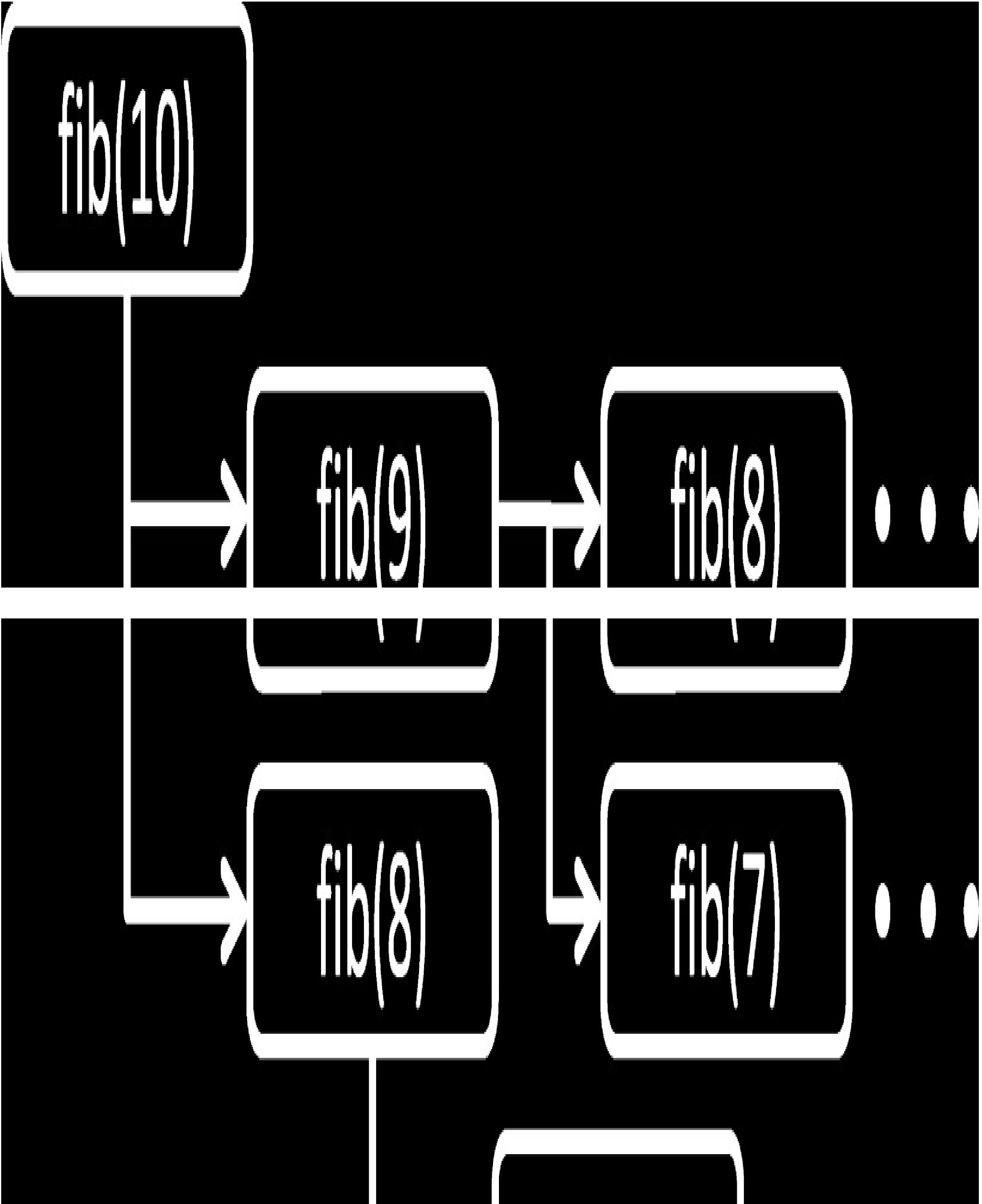
失控崩溃了。

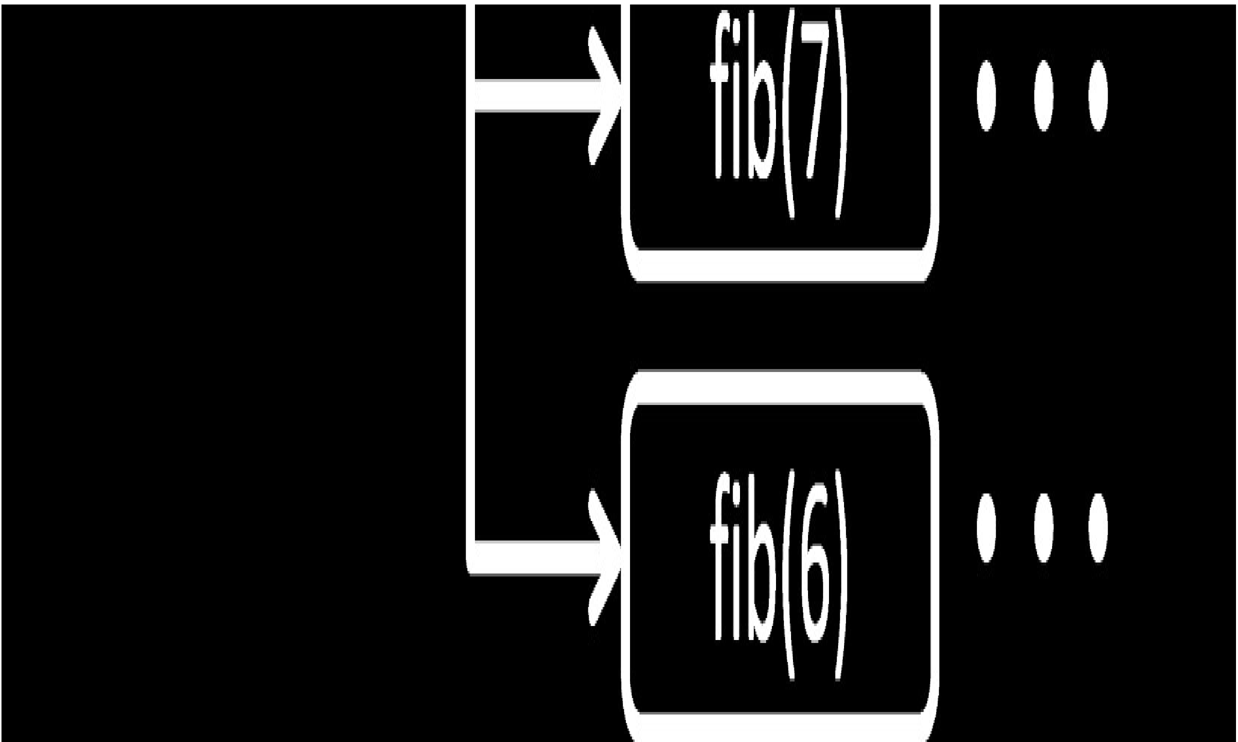
fact递归的过程

我们再来试试编写计算斐波那契数列的函数int fib(int n)。斐波那契数列的定义是 $a_0=0$、 $a_1=1$

以及 $a_n = a_{n-1} + a_{n-2}$ ($n>1$)。这里，初项的条件就对应了递归的终止条件。数列的定义直接写成函数

就可以了。


fib(10)
fib(9)
fib(8)
fib(8)
fib(7)




```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

1

实际使用这个函数时，即使是求fib(40)这样的 *n* 较小时的结果，也要花费相当长的时间。这是

2

因为这个函数在递归时，会像下图一样按照指数级别扩展开来。

3

4

fib(10) 递归的过程

在斐波那契数列中，如果fib(n)的 *n* 是一定的，无论多少次调用都会得到同样的结果。因此如果

5

计算一次之后，用数列将结果存储起来，便可优化之后的计算。（上图中）fib(10)被调用时同

样的 *n* 被计算了很多次，因此可以获得很大的优化空间。这种方法是出于记忆化搜索或者动态规

划的想法，之后我们会介绍。

6

```
int memo[MAX_N + 1];

int fib(int n) {

if (n <= 1) return n;
```

7

```
if (memo[n] != 0) return memo[n];

return memo[n] = fib(n - 1) + fib(n - 2);

}
```

8

2.1.2 栈

栈（Stack）是支持push和pop两种操作的数据结构。push是在栈的顶端放入一组数据的操作。反

之，pop是从其顶端取出一组数据的操作。因此，最后进入栈的一组数据可以最先被取出（这种

9

行为被叫做LIFO: Last In First Out，即后进先出）。

通过使用数组或者列表等结构可以很容易实现栈，不过C++、Java等程序语言的标准库已经为我

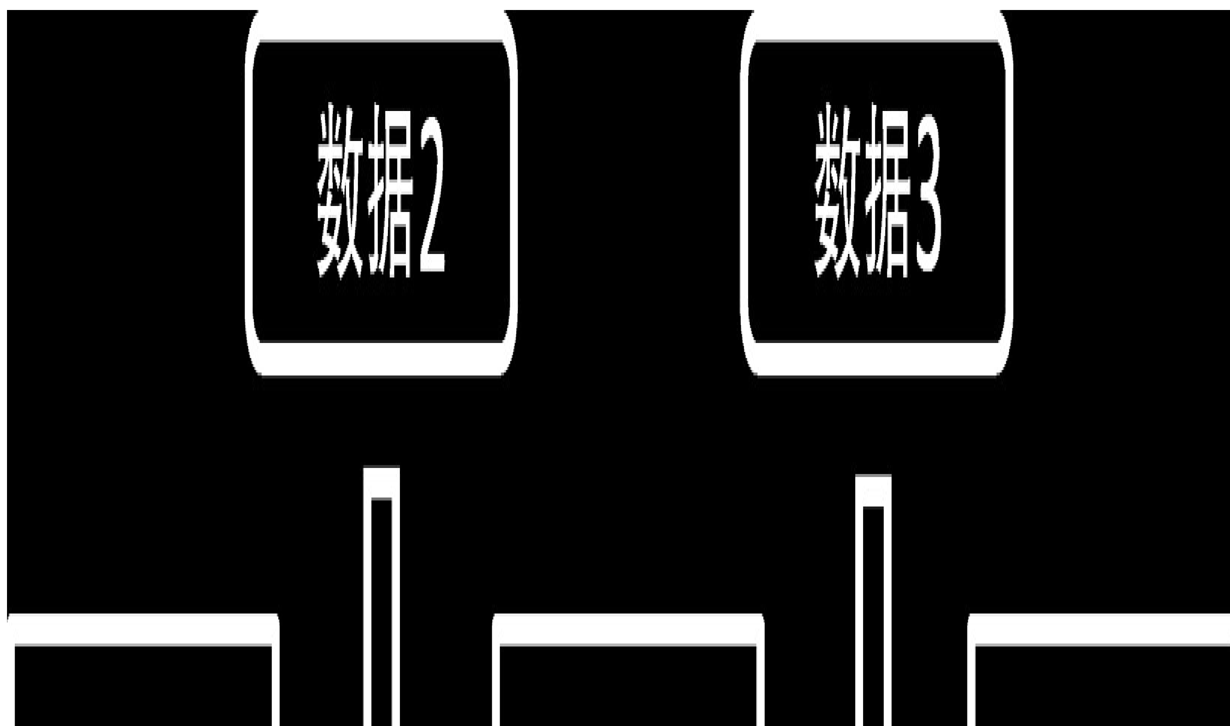
们准备好了这一常用结构，在比赛中需要时不妨使用它们。C++的标准库中，stack::pop完成

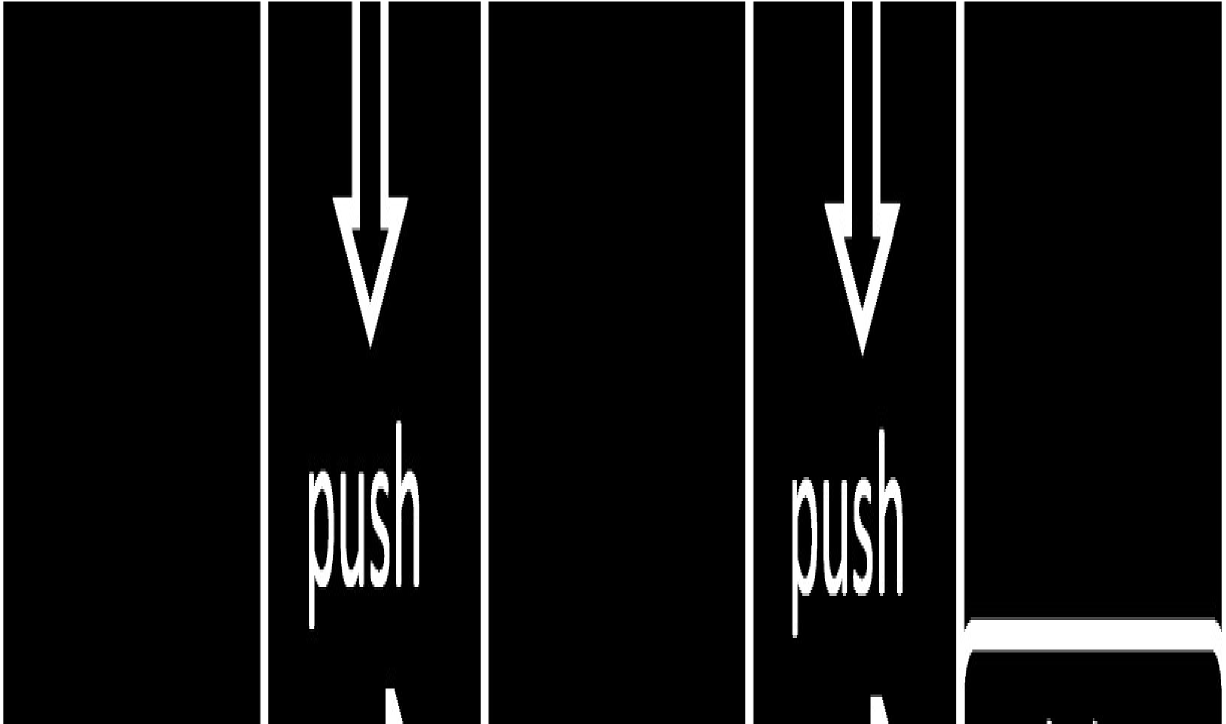
10

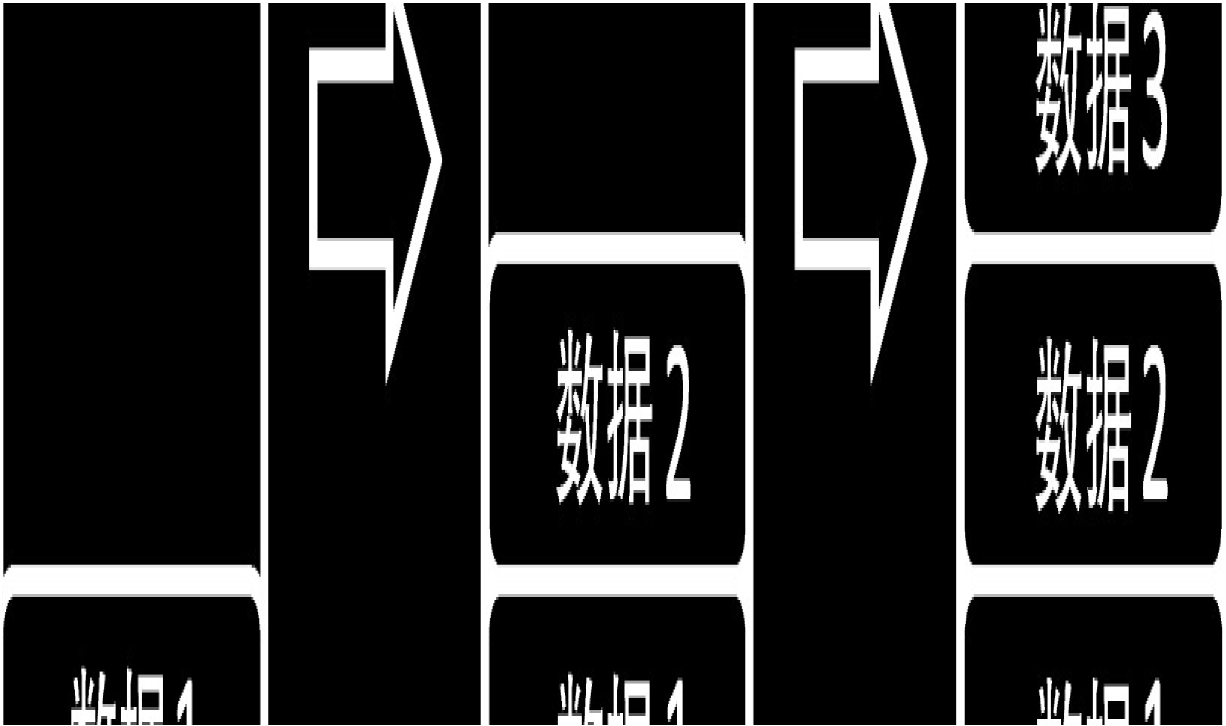
的仅仅是移除最顶端的数据。如果要访问最顶端的数据，需要使用stack::top函数（这个操作

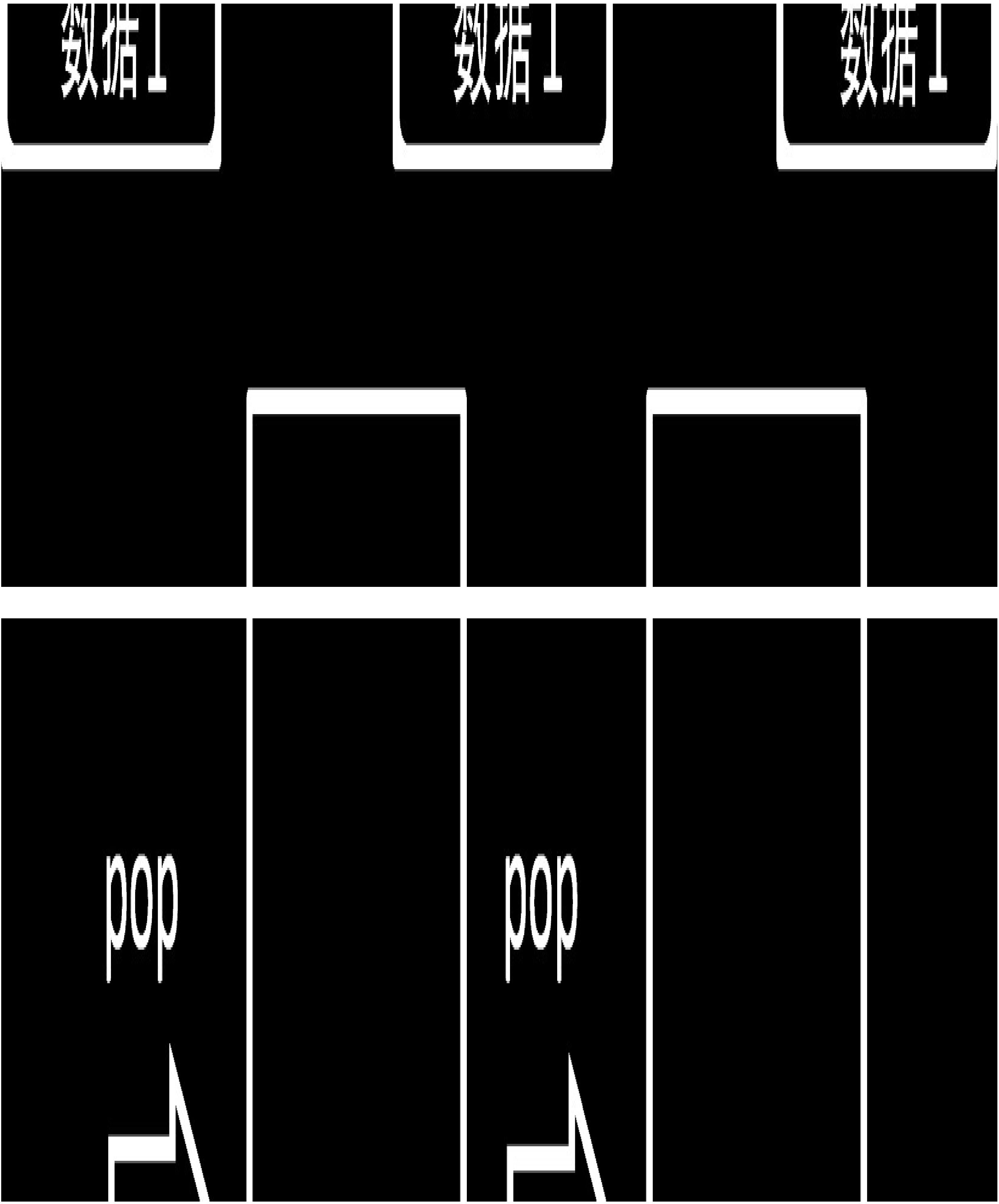
通常也被称为peek）。

11

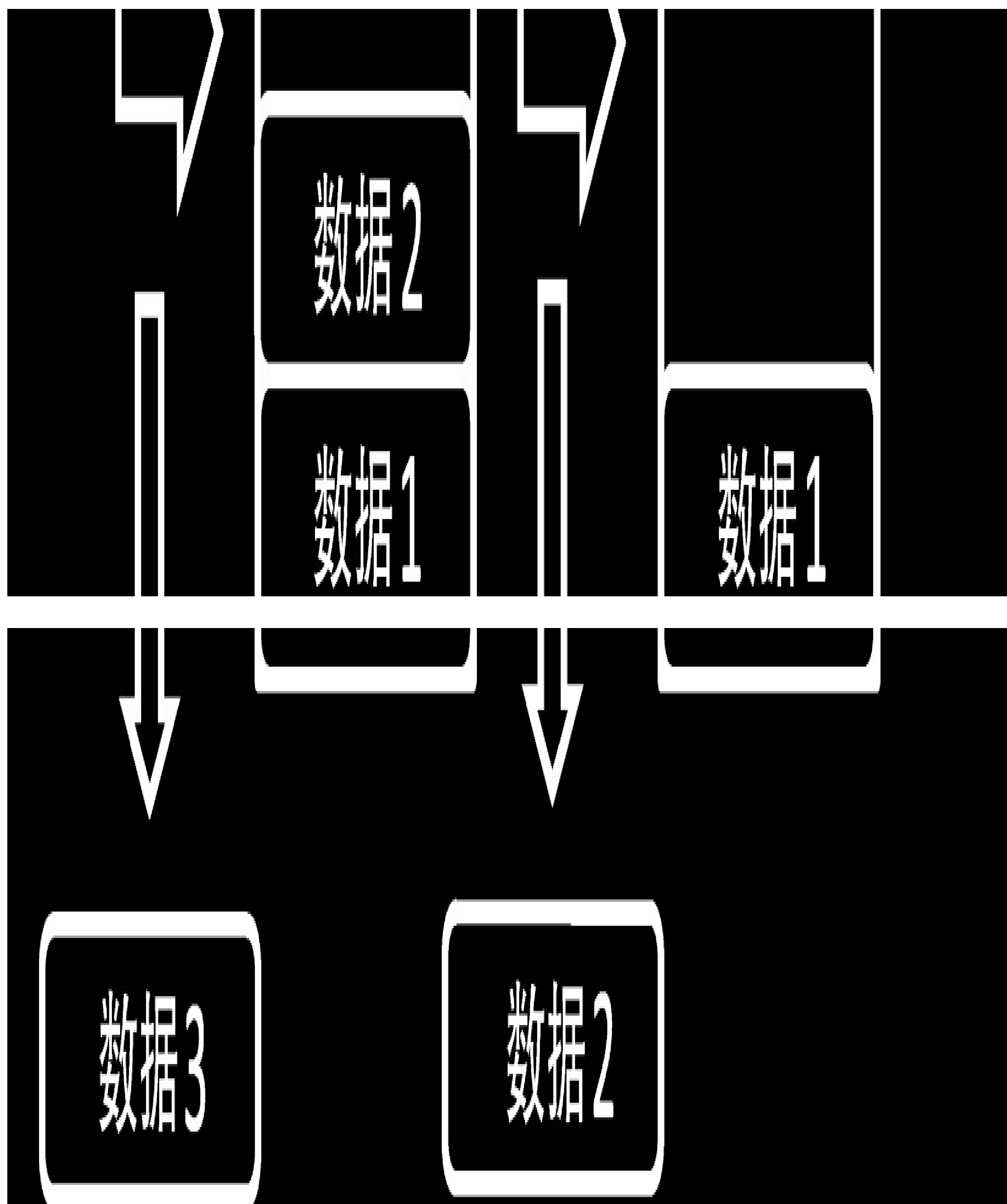



push
push


数据 3
数据 2
数据 2



数据1
数据1
数据1
pop
pop



栈的操作

函数调用的过程是通过使用栈实现的。因此，递归函数的递归过程也可以改用栈上的操作来实现。

现实中需要如此改写的场合并不多，不过作为使用栈的练习试试看也是不错的。以下是使用

stack的例子：

```
#include <stack>

#include <cstdio>

using namespace std;

int main() {

stack<int> s; // 声明存储int类型数据的栈

s.push(1); // {} → {1}

s.push(2); // {1} → {1,2}

s.push(3); // {1,2} → {1,2,3}

printf("%d\n", s.top()); // 3

s.pop(); // 从栈顶移除 {1,2,3}→{1,2}

printf("%d\n", s.top()); // 2

s.pop(); // {1,2} → {1}

printf("%d\n", s.top()); // 1

s.pop(); // {1} → {}

return 0;

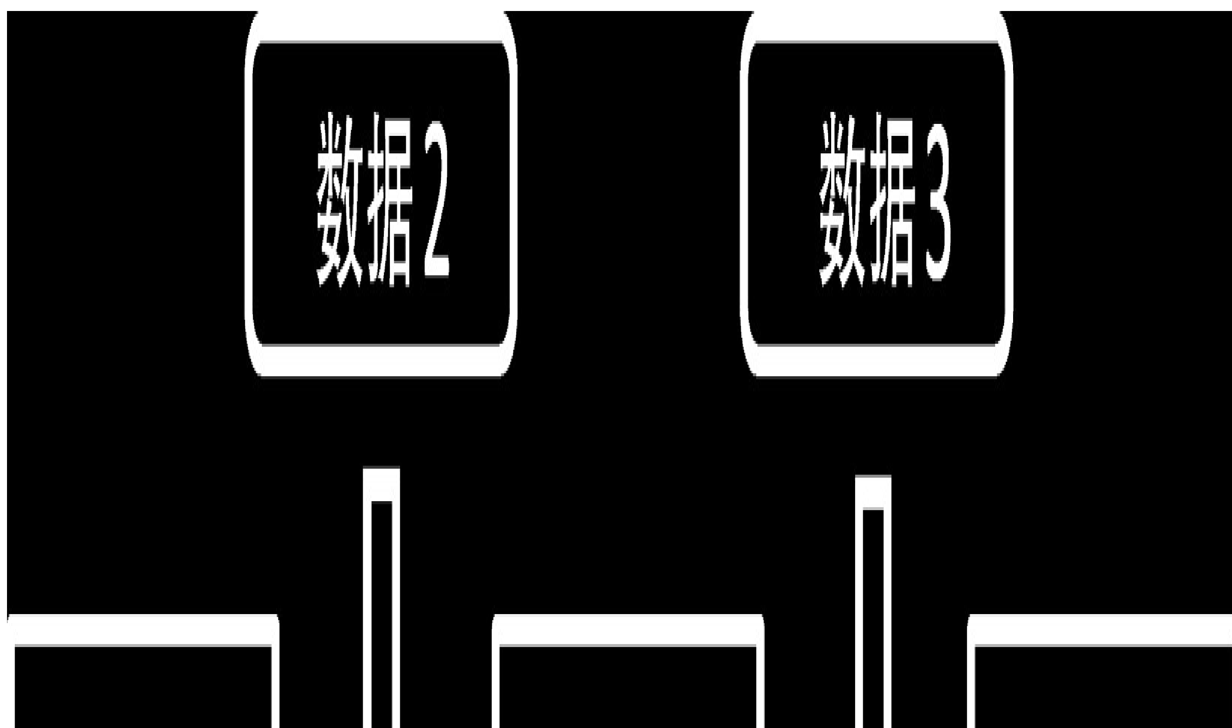
}
```

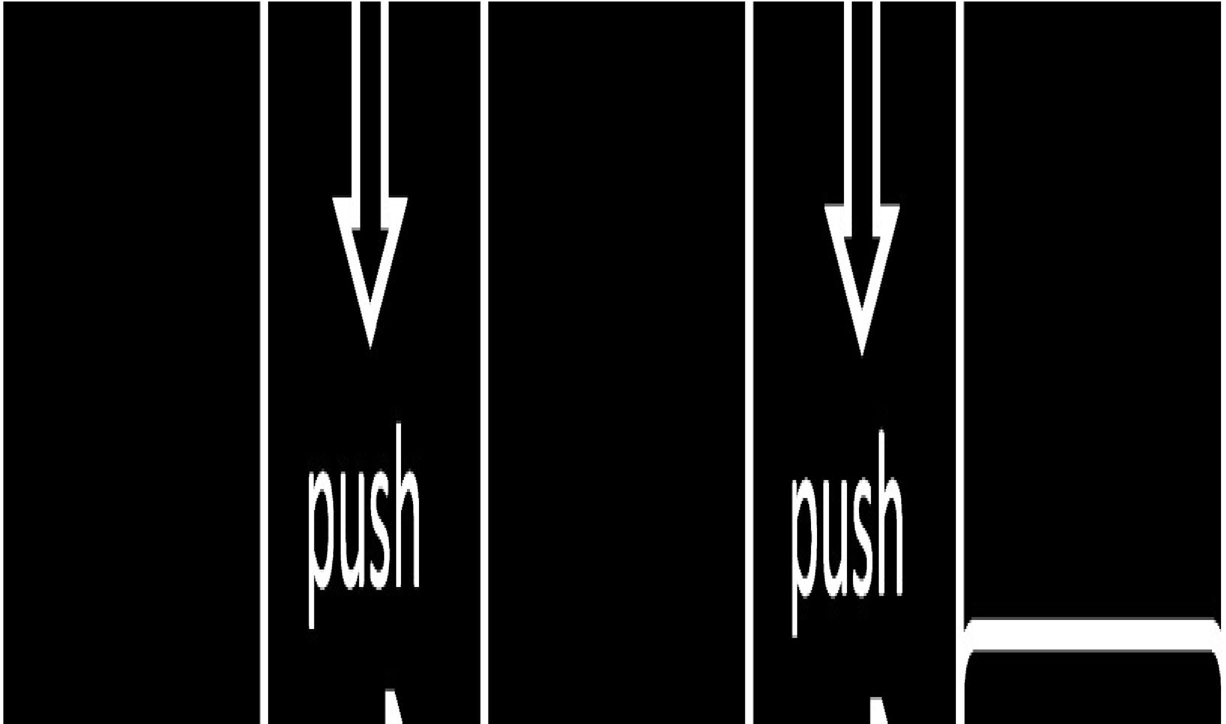
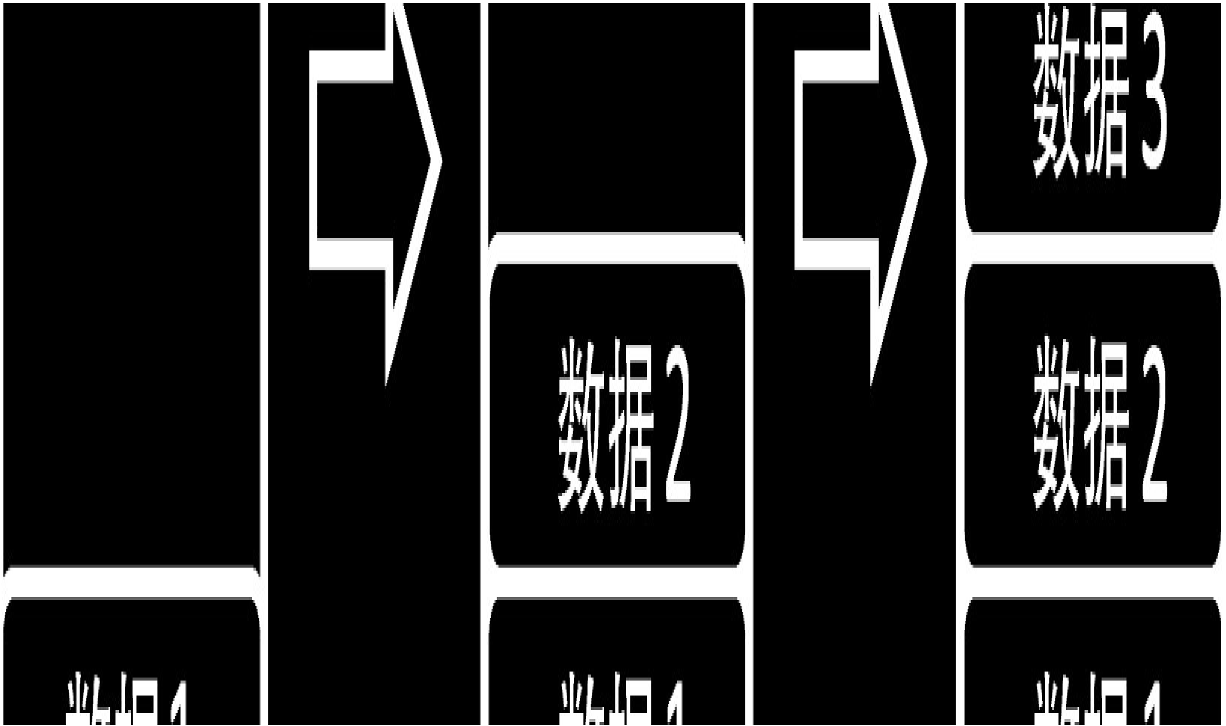
2.1.3 队列

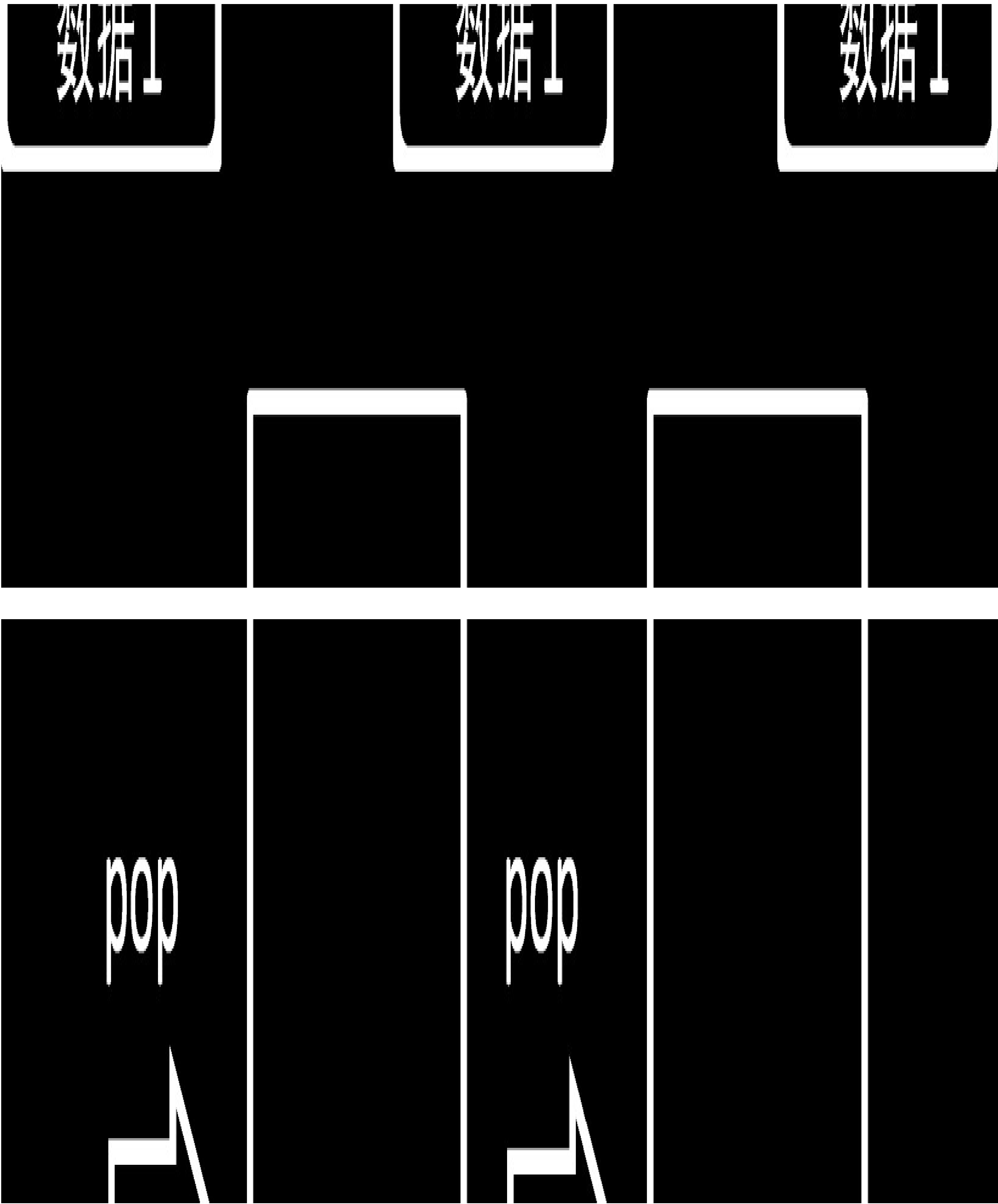
队列（Queue）与栈一样支持push和pop两个操作。但与栈不同的是，pop完成的不是取出最顶端

的元素，而是取出最底端的元素。也就是说最初放入的元素能够最先被取出（这种行为被叫做FIFO:

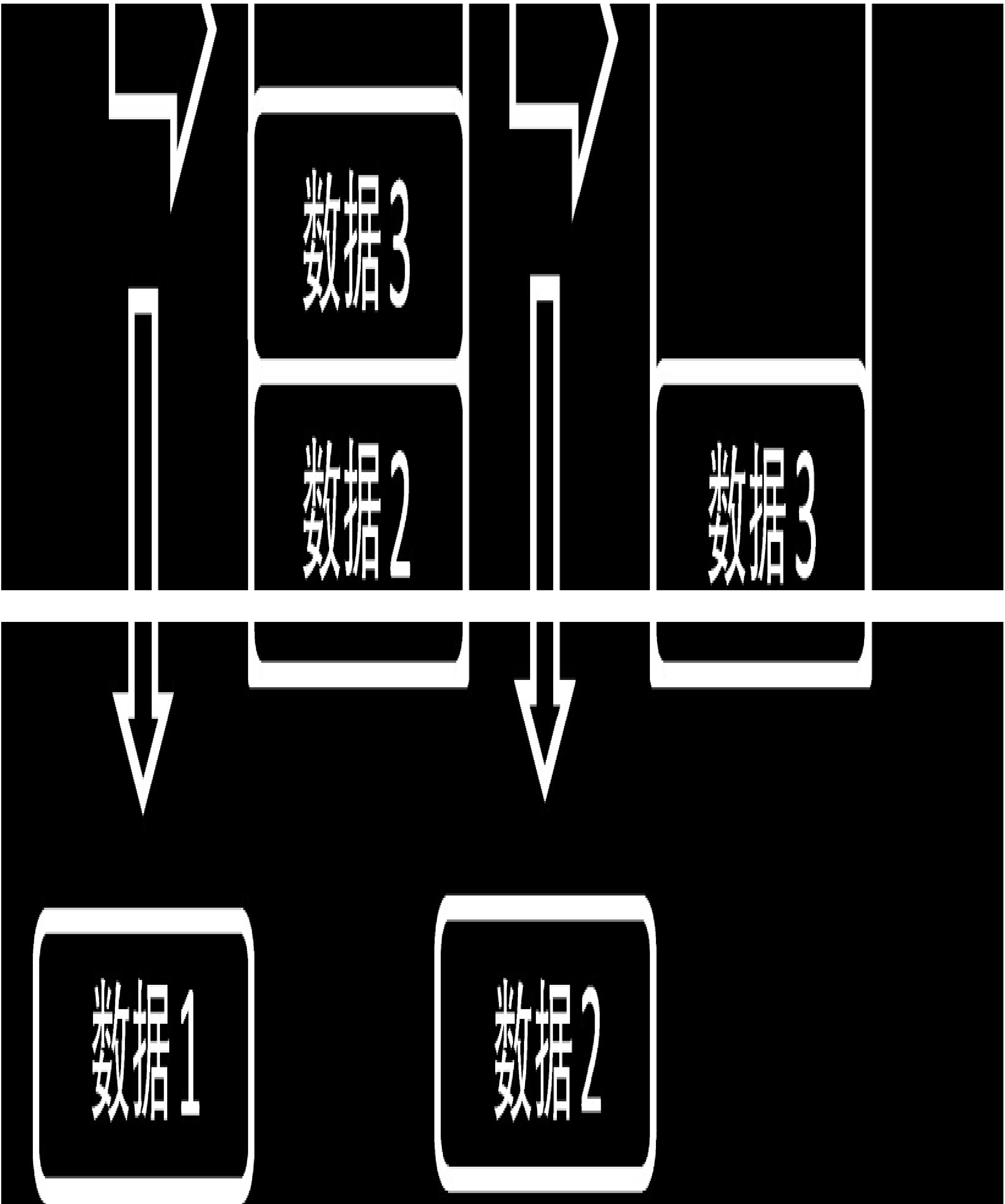
First In First Out，即先进先出）。




push
push

数据 3
数据 2
数据 2



数据1
数据1
数据1
pop
pop



1

2

3

4

5

队列的操作

如同栈一样，C++、Java等的标准库也预置了队列。Java与C++中的函数的名称与用途稍有不同，

因此使用时要注意。此外，在C++中queue::front是用来访问最底端数据的函数。以下是使用

6

queue的例子：

#include <queue>

#include <cstdio>

7

using namespace std;

int main() {

queue<int> que; // 声明存储int类型数据的队列

que.push(1); // {} → {1}

8

que.push(2); // {1} → {1,2}

que.push(3); // {1,2} → {1,2,3}

```
printf("%d\n", que.front()); // 1

que.pop(); // 从队尾移除 {1,2,3}→{2,3}

printf("%d\n", que.front()); // 2
```

9

```
que.pop(); // {2,3} → {3}

printf("%d\n", que.front()); // 3

que.pop(); // {3} → {}

return 0;

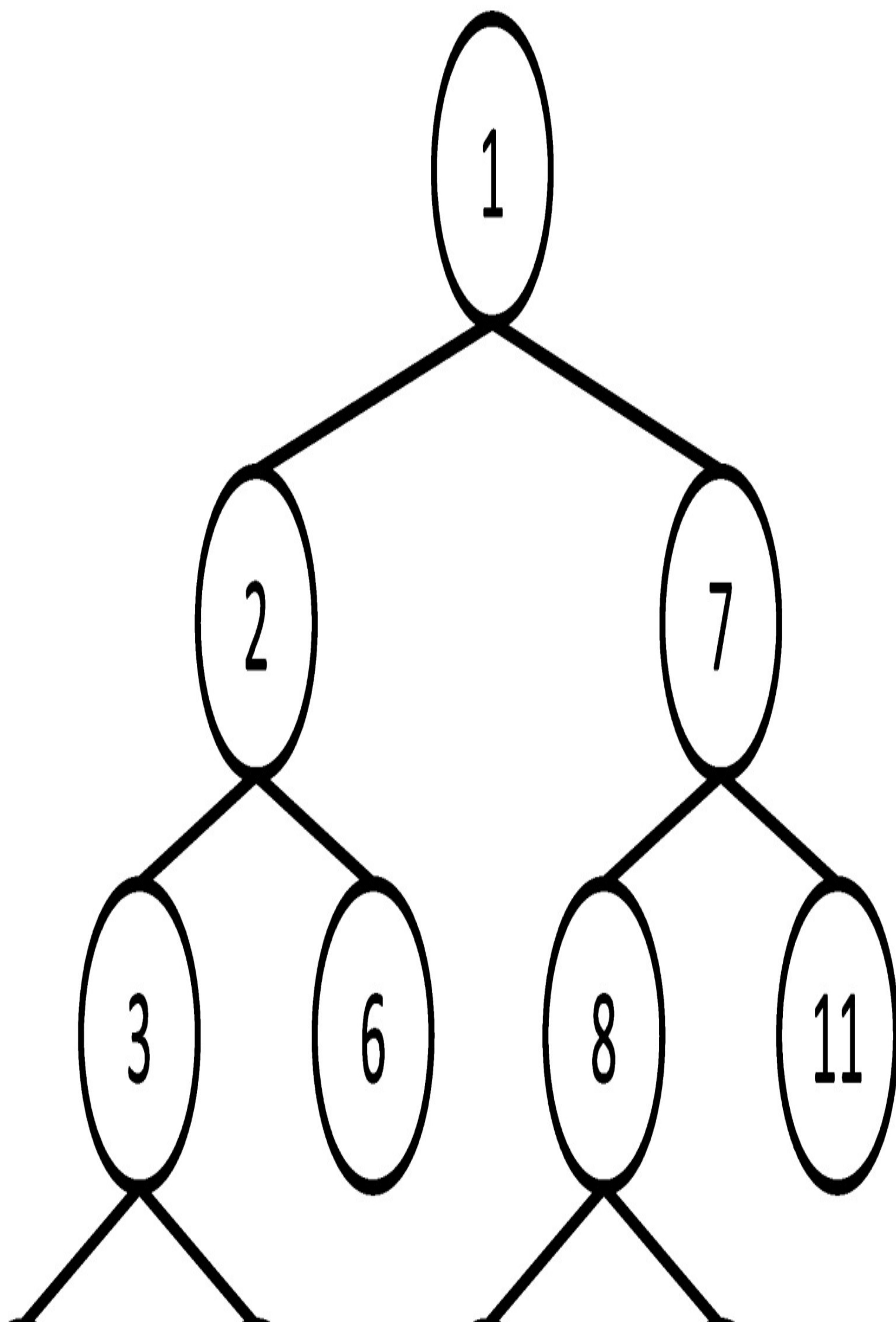
}
```

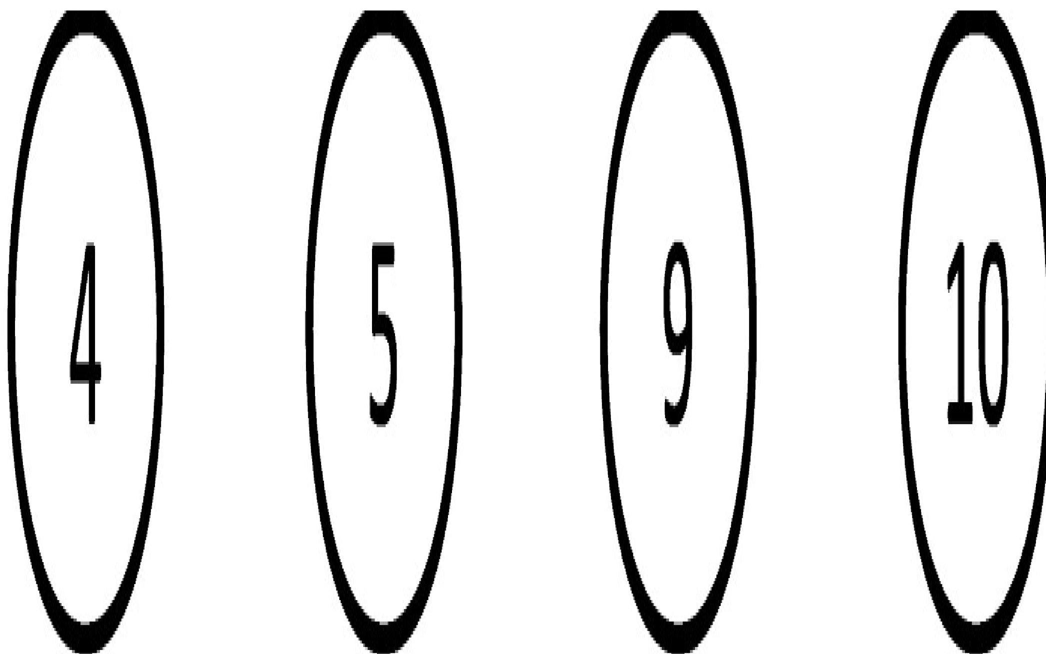
10

2.1.4 深度优先搜索

11

深度优先搜索（DFS，Depth-First Search）是搜索的手段之一。它从某个状态开始，不断地转移


1
2
7
3
6
8
11





状态直到无法转移，然后回退到前一步的状态，继续转移到其他状态，如此不断重复，直至找到

最终的解。例如求解数独，首先在某个格子内填入适当的数字，然后再继续在下一个格子内填入

数字，如此继续下去。如果发现某个格子无解了，就放弃前一个格子上选择的数字，改用其他可

行的数字。根据深度优先搜索的特点，采用递归函数实现比较简单。

状态转移的顺序

我们来试着解答一下下面的题目：

部分和问题

给定整数 a_1、a_2、…、a_n，判断是否可以从中选出若干数，使它们的和恰好为 k 。

限制条件

□ $1 \le n \le 20$

□ □$108 \le a_i \le 108$

□ □$108 \le k \le 108$

样例 1

输入

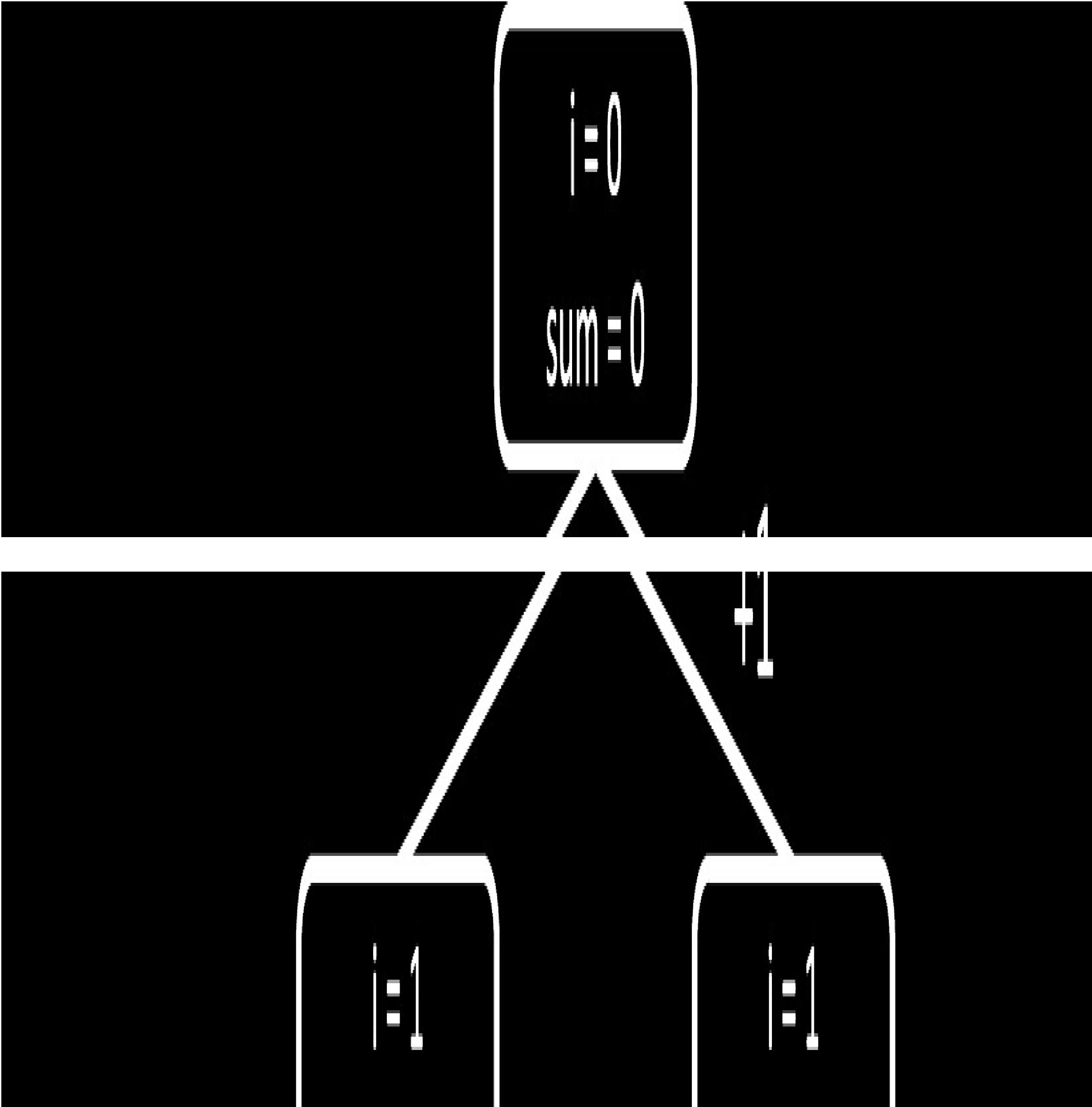
n=4

a={1,2,4,7}

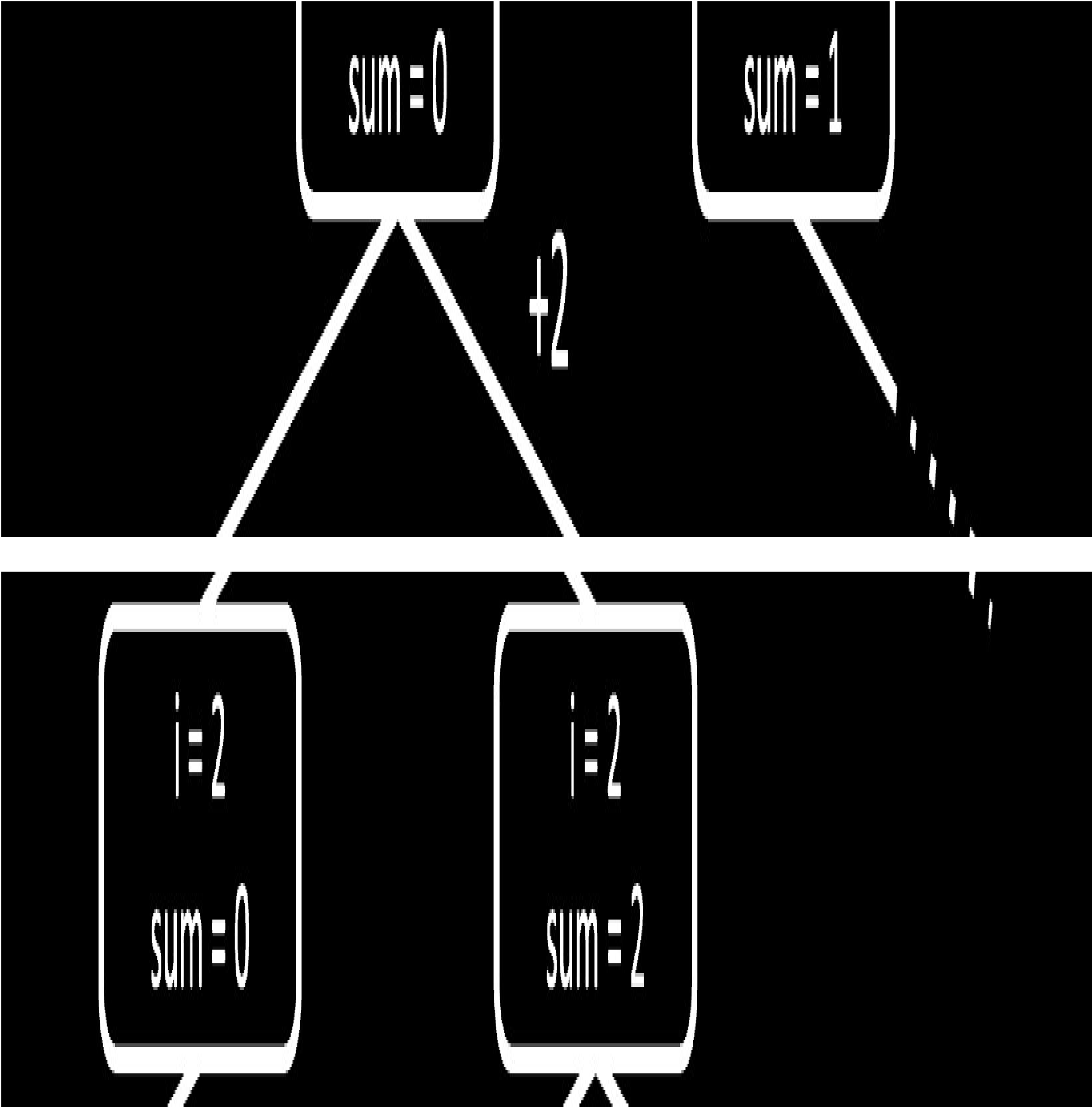
k=13

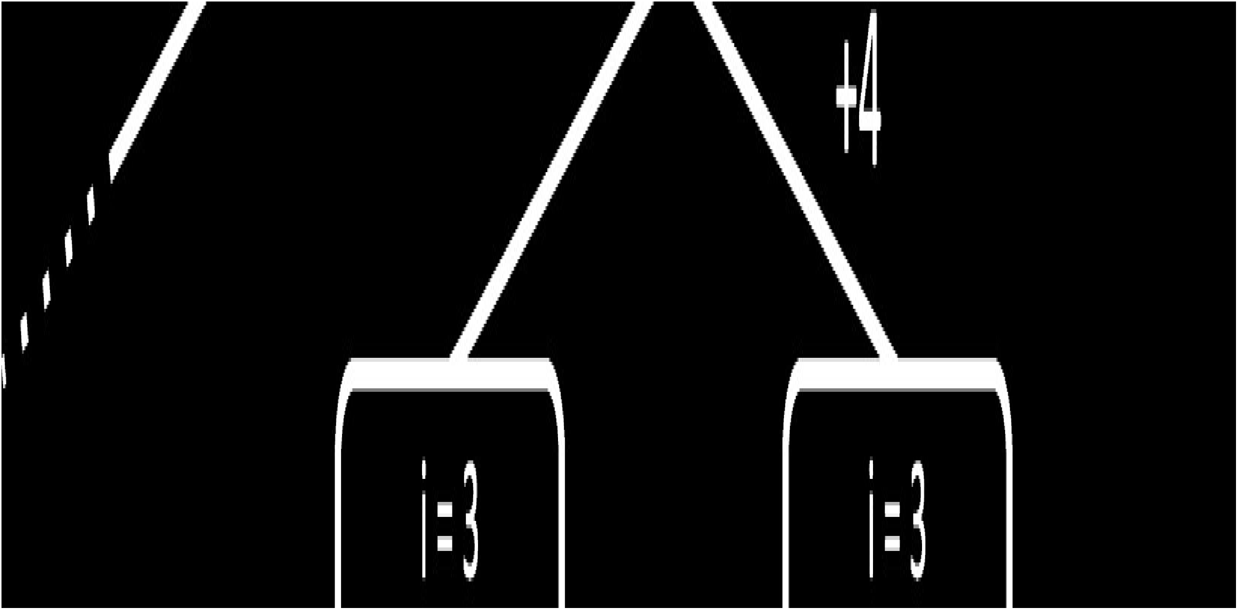
输出

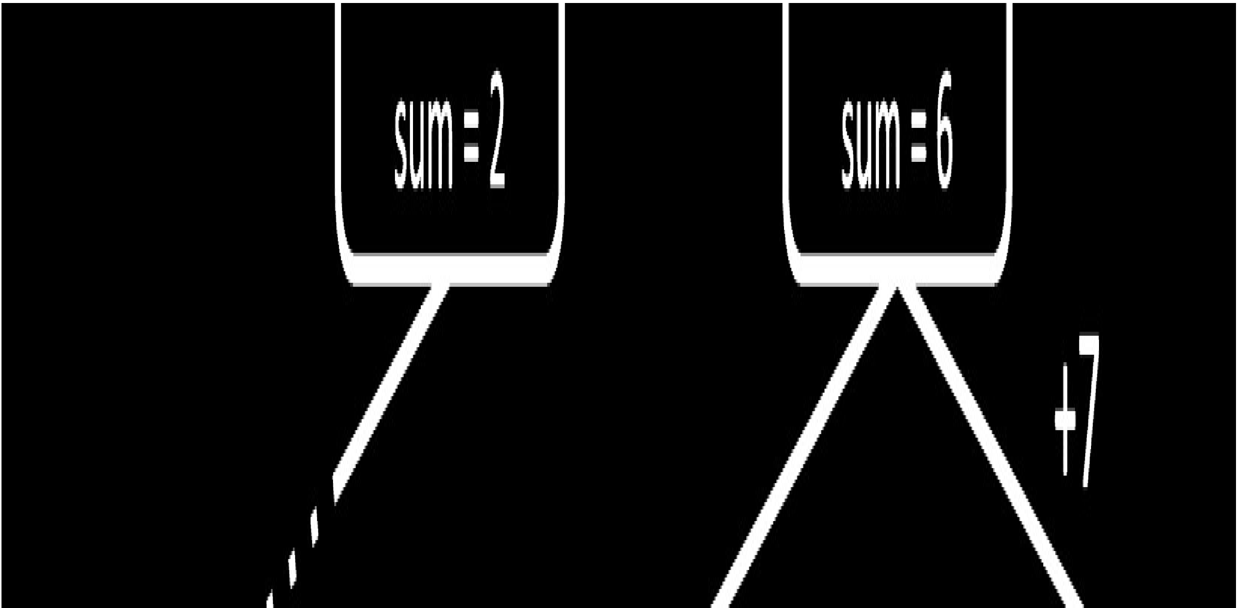
Yes (13 = 2 + 4 + 7)

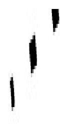


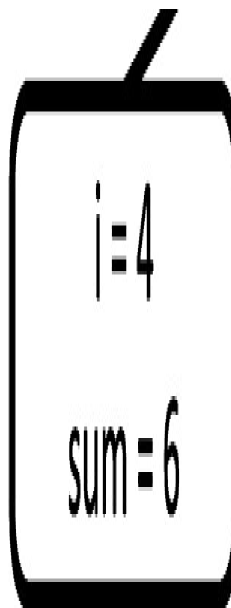
i=0
sum=0
+1
i=1
i=1

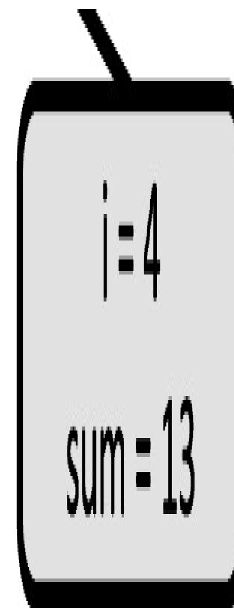

sum = 0
sum = 1
+2
i = 2
sum = 0
i = 2
sum = 2


+4
i = 3
i = 3


sum = 2
sum = 6
+7







样例 2

1

输入

n=4

a={1,2,4,7}

k=15

2

输出

No

3

从 a 1开始按顺序决定每个数加或不加，在全部 n 个数都决定后再判断它们的和是不是 k 即可。因为

状态数是2^{n+1}，所以复杂度是 $O(2^n)$。如何实现这个搜索，请参见下面的代码。注意 a 的下标与题

目描述中的下标偏移了1。在程序中使用的是0起始的下标规则，题目描述中则是1开始的，这一

4

点要注意避免搞混。

5

6

7

8

状态转移的样子

// 输入

int a[MAX_N];

9

int n, k;

// 已经从前i项得到了和sum，然后对于i项之后的进行分支

bool dfs(int i, int sum) {

// 如果前n项都计算过了，则返回sum是否与k相等

10

if (i == n) return sum == k;

// 不加上a[i]的情况

```
if (dfs(i + 1, sum)) return true;
```

11

```
// 加上a[i]的情况
```




```
if (dfs(i + 1, sum + a[i])) return true;
// 无论是否加上a[i]都不能凑成k就返回false
return false;
}
void solve() {
if (dfs(0, 0)) printf("Yes\n");
else printf("No\n");
}
```

深度优先搜索从最开始的状态出发，遍历所有可以到达的状态。由此可以对所有的状态进行操作，

或者列举出所有的状态。

Lake Counting （POJ No.2386）

有一个大小为 $N \times M$ 的园子，雨后积起了水。八连通的积水被认为是连接在一起的。请求出

园子里总共有多少水洼？（八连通指的是下图中相对 W 的*的部分）



限制条件

□ $N, M \leq 100$

样例

输入

N=10, M=12

园子如下图（'W'表示积水，'.'表示没有积水）

W........WW.

.WWW.....WWW

....WW...WW.

.........WW.

.........W..

..W......W..

.W.W.....WW.

W.W.W.....W.

.W.W......W.

..W.......W.

输出

3

从任意的W开始，不停地把邻接的部分用'.'代替。1次DFS后与初始的这个W连接的所有W就都被替

1

换成了'.'，因此直到图中不再存在W为止，总共进行DFS的次数就是答案了。8个方向共对应了8种

状态转移，每个格子作为DFS的参数至多被调用一次，所以复杂度为 $O(8\times N\times M)=O(N\times M)$。

// 输入

2

int N, M;

char field[MAX_N][MAX_M + 1]; // 园子

// 现在位置(x,y)

void dfs(int x, int y) {

3

// 将现在所在位置替换为.

field[x][y] = '.';

// 循环遍历移动的8个方向

for (int dx = -1; dx <= 1; dx++) {

```
for (int dy = -1; dy <= 1; dy++) {
```

4

```
// 向x方向移动dx，向y方向移动dy，移动的结果为（nx,ny）

int nx = x + dx, ny = y + dy;

// 判断(nx,ny)是不是在园子内，以及是否有积水

if (0 <= nx && nx < N && 0 <= ny && ny < M && field[nx][ny] == 'W')
dfs(nx, ny);

}
```

5

```
}

return ;

}

void solve() {
```

6

```
int res = 0;

for (int i = 0; i < N; i++) {

for (int j = 0; j < M; j++) {

if (field[i][j] == 'W') {

// 从有W的地方开始dfs
```

7

```
dfs(i, j);
```

```
res++;
}
}
}
printf("%d\n", res);
```

8

```
}
```

2.1.5 宽度优先搜索

9

宽度优先搜索（BFS，Breadth-First Search）也是搜索的手段之一。它与深度优先搜索类似，从某

个状态出发探索所有可以到达的状态。

与深度优先搜索的不同之处在于搜索的顺序，宽度优先搜索总是先搜索距离初始状态近的状态。

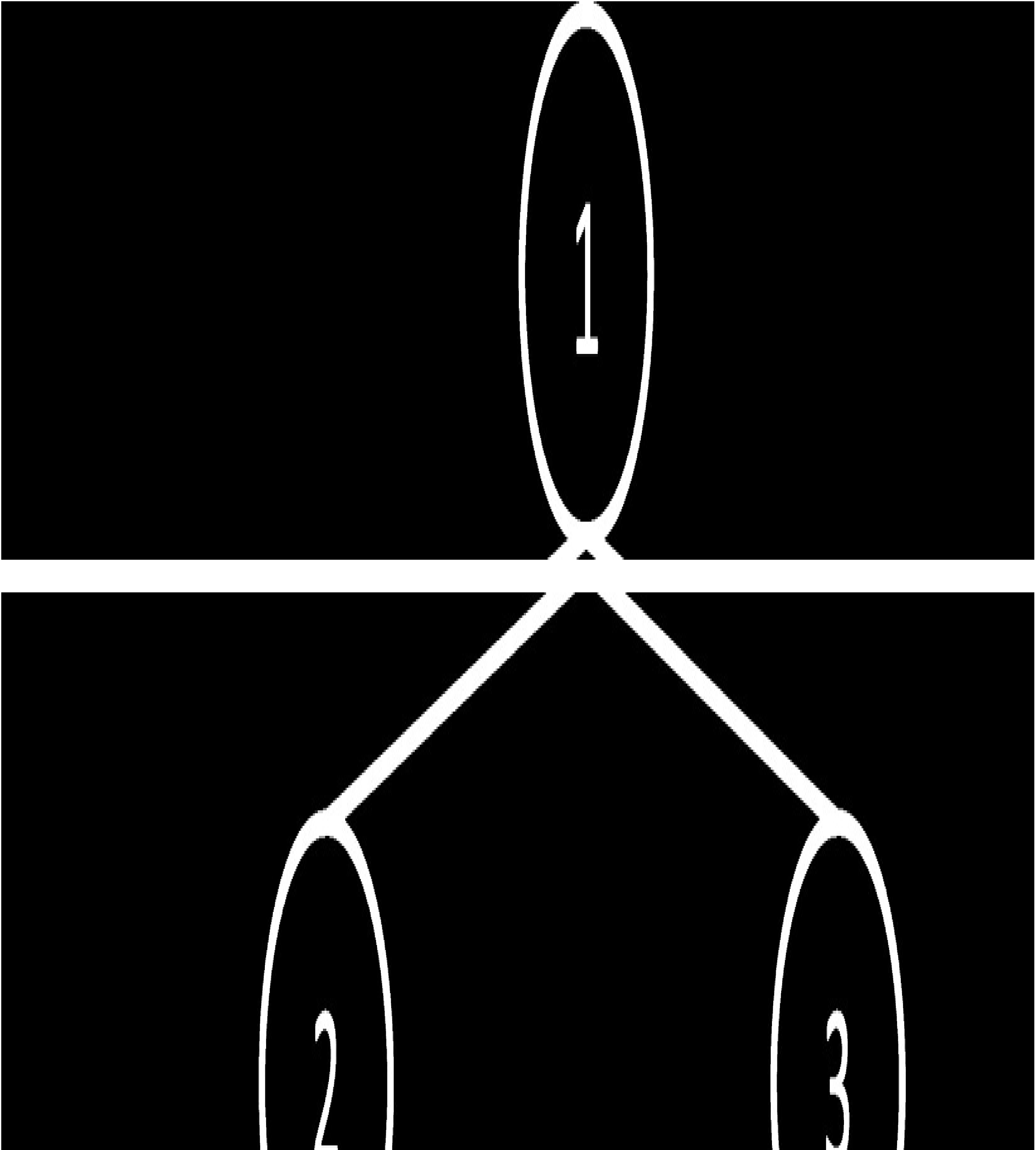
10

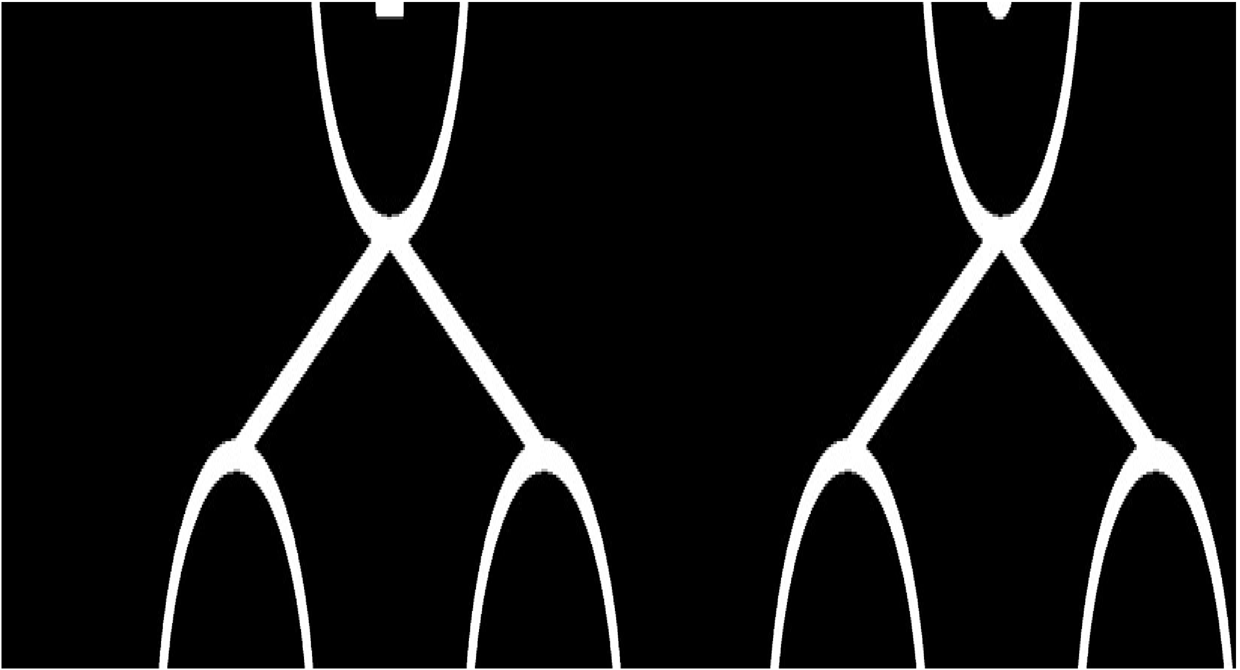
也就是说，它是按照开始状态→只需1次转移就可以到达的所有状态→只需2次转移就可以到达的

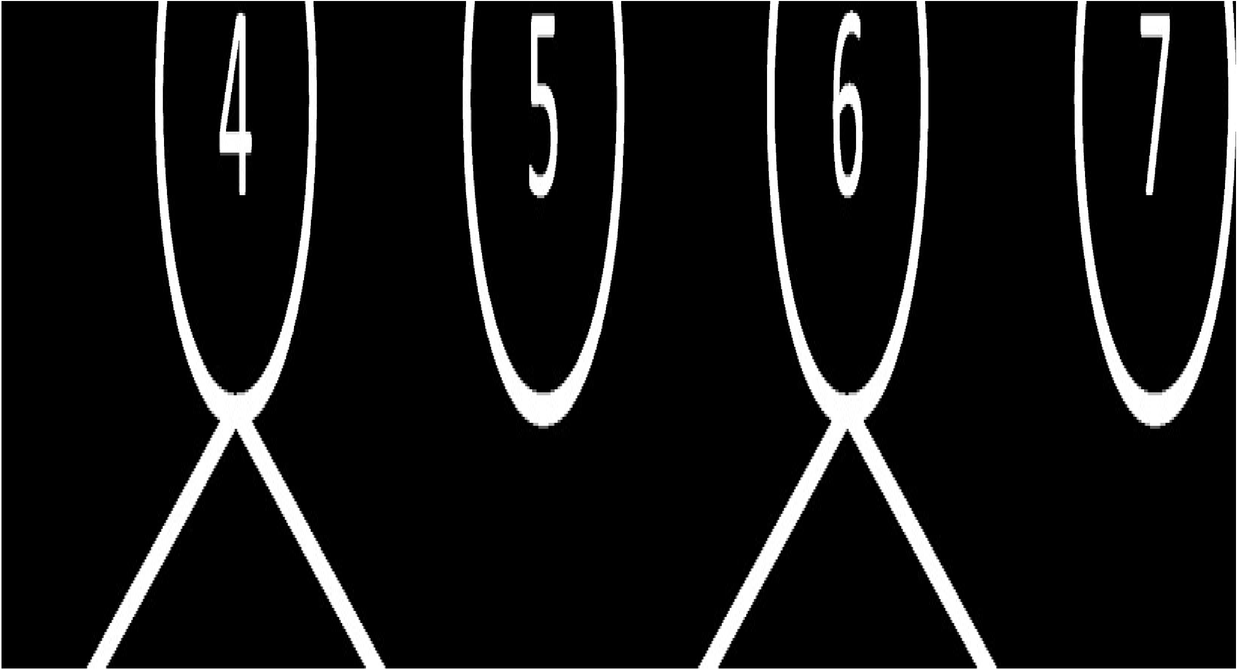
所有状态→……这样的顺序进行搜索。对于同一个状态，宽度优先搜索只经过一次，因此复杂度

为 *O* (状态数×转移的方式)。

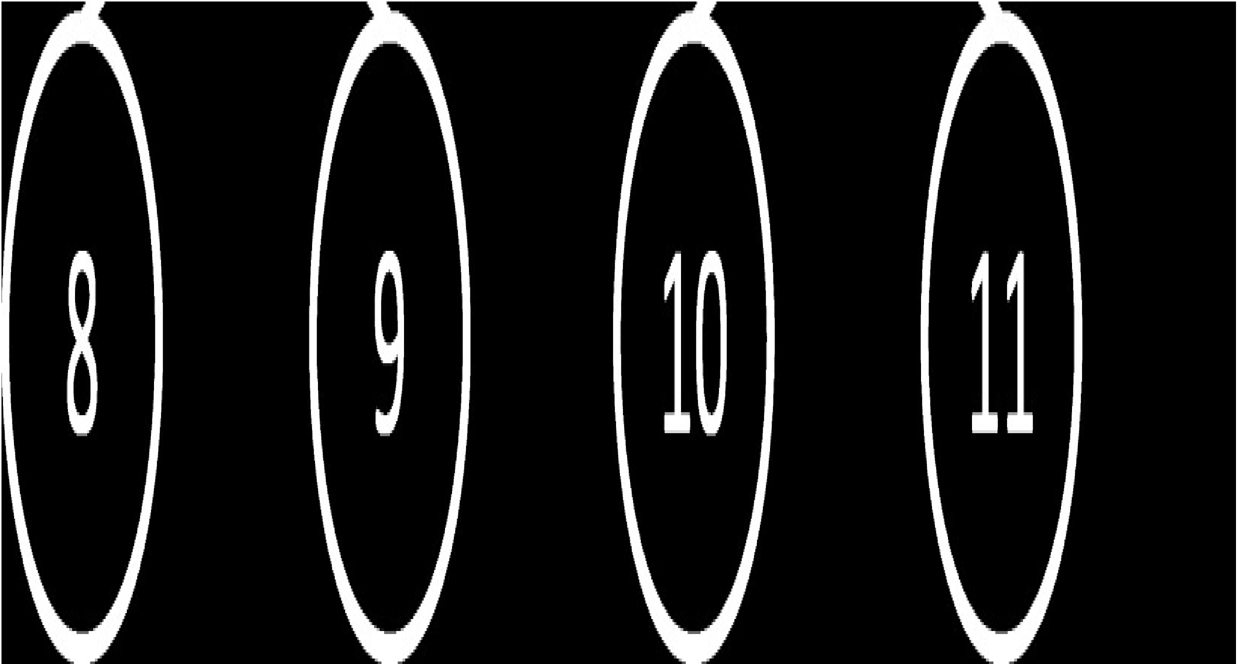
11


1
2
3





4
5
6
7





状态转移的顺序

深度优先搜索（隐式地）利用了栈进行计算，而宽度优先搜索则利用了队列。搜索时首先将初始

状态添加到队列里，此后从队列的最前端不断取出状态，把从该状态可以转移到的状态中尚未访

问过的部分加入队列，如此往复，直至队列被取空或找到了问题的解。通过观察这个队列，我们

可以就知道所有的状态都是按照距初始状态由近及远的顺序被遍历的。

迷宫的最短路径

给定一个大小为 $N \times M$ 的迷宫。迷宫由通道和墙壁组成，每一步可以向邻接的上下左右四格

的通道移动。请求出从起点到终点所需的最小步数。请注意，本题假定从起点一定可以移动

到终点。

限制条件

□ $N, M \leq 100$

样例

输入

N=10, M=10（迷宫如下图所示。'#'，'.'，'S'，'G'分别表示墙壁、通道、起点和终点）

```
#S######.#
......#..#
.#.##.##.#
.#........
##.##.####
....#....#
.#######.#
....#.....
.####.###.
....#...G#
```

输出

1

22

宽度优先搜索按照距开始状态由近及远的顺序进行搜索，因此可以很容易地用来求最短路径、最

少操作之类问题的答案。这个问题中，状态仅仅是目前所在位置的坐标，因此可以构造成pair

2

或者编码成int来表达状态。当状态更加复杂时，就需要封装成一个类来表示状态了。转移的方

式为四方向移动，状态数与迷宫的大小是相等的，所以复杂度是 $O(4\times N\times M)=O(N\times M)$。

宽度优先搜索中，只要将已经访问过的状态用标记管理起来，就可以很好地做到由近及远的搜索。 **3**

这个问题中由于要求最短距离，不妨用d[N][M]数组把最短距离保存起来。初始时用充分大的常

数INF来初始化它，这样尚未到达的位置就是INF，也就同时起到了标记的作用。

4

虽然到达终点时就会停止搜索，可如果继续下去直到队列为空的话，就可以计算出到各个位置的

最短距离。此外，如果搜索到最后，d依然为INF的话，便可得知这个位置就是无法从起点到达的

位置。

5

在今后的程序中，使用像INF这样充分大的常数的情况还很多。不把INF当作例外，而是直接参

与普通运算的情况也很常见。这种情况下，如果INF过大就可能带来溢出的危险。

假设INF=231-1。例如想用d[nx][ny]=min(d[nx][ny], d[x][y]+1)来更新d[nx][ny]，就会

6

发生INF+1=-231的情况。这一问题中d[x][y]总不等于INF，所以没有问题。但是为了防止这样

的问题，一般会将INF设为放大2~4倍也不会溢出的大小（可参考2.5节Floyd-Warshall算法等）。

因为要向4个不同方向移动，用dx[4]和dy[4]两个数组来表示四个方向向量。这样通过一个循

7

环就可以实现四方向移动的遍历。

```
const int INF = 100000000;

// 使用pair表示状态时，使用typedef会更加方便一些
```

8

```
typedef pair<int, int> P;

// 输入

char maze[MAX_N][MAX_M + 1]; // 表示迷宫的字符串的数组
```

```
int N, M;

9

int sx, sy; // 起点坐标

int gx, gy; // 终点坐标

int d[MAX_N][MAX_M]; // 到各个位置的最短距离的数组

10

// 4个方向移动的向量

int dx[4] = {1, 0, -1, 0}, dy[4] = {0, 1, 0, -1};

// 求从(sx, sy)到(gx, gy)的最短距离

// 如果无法到达，则是INF

int bfs() {

11
```



```
queue<P> que;

// 把所有的位置都初始化为INF

for (int i = 0; i < N; i++)

for (int j = 0; j < M; j++) d[i][j] = INF;

// 将起点加入队列，并把这一地点的距离设置为0

que.push(P(sx, sy));

d[sx][sy] = 0;
```

```
// 不断循环直到队列的长度为0

while (que.size()) {

// 从队列的最前端取出元素

P p = que.front(); que.pop();

// 如果取出的状态已经是终点，则结束搜索

if (p.first == gx && p.second == gy) break;

// 四个方向的循环

for (int i = 0; i < 4; i++) {

// 移动之后的位置记为(nx, ny)

int nx = p.first + dx[i], ny = p.second + dy[i];

// 判断是否可以移动以及是否已经访问过（d[nx][ny]!=INF即为已经访问过）

if (0 <= nx && nx < N && 0 <= ny && ny < M && maze[nx][ny] != '#' && d[nx][ny] == INF) {

// 可以移动的话，则加入到队列，并且到该位置的距离确定为到p的距离+1

que.push(P(nx, ny));

d[nx][ny] = d[p.first][p.second] + 1;

}

}

}

return d[gx][gy];
```

```
}

void solve() {

int res = bfs();

printf("%d\n", res);

}
```

宽度优先搜索与深度优先搜索一样，都会生成所有能够遍历到的状态，因此需要对所有状态进行

处理时使用宽度优先搜索也是可以的。但是递归函数可以很简短地编写，而且状态的管理也更简

单，所以大多数情况下还是用深度优先搜索实现。反之，在求取最短路时深度优先搜索需要反复

经过同样的状态，所以此时还是使用宽度优先搜索为好。

宽度优先搜索会把状态逐个加入队列，因此通常需要与状态数成正比的内存空间。反之，深度优

先搜索是与最大的递归深度成正比的。一般与状态数相比，递归的深度并不会太大，所以可以认

为深度优先搜索更加节省内存。

此外，也有采用与宽度优先搜索类似的状态转移顺序，并且注重节约内存占用的迭代加深深度优

先搜索（IDDFS，Iterative Deepening Depth-First Search）。IDDFS是一种在最开始将深度优先搜索

的递归次数限制在1次，在找到解之前不断增加递归深度的方法。这种方法会在4.5节详细介绍。

1

2.1.6 特殊状态的枚举

虽然生成可行解空间多数采用深度优先搜索，但在状态空间比较特殊时其实可以很简短地实现。

比如，C++的标准库中提供了next_permutation这一函数，可以把 n 个元素共 n !种不同的排列生

2

成出来。又或者，通过使用位运算，可以枚举从 n 个元素中取出 k 个的共 k

C 种状态或是某个集合

n

中的全部子集等。3.2节将介绍如何利用位运算枚举状态。

```
bool used[MAX_N];

int perm[MAX_N];
```

3

```
// 生成{0,1,2,3,4,...,n-1}的n!种排列

void permutation1(int pos, int n) {

if (pos == n) {
```

4

```
/*

* 这里编写需要对perm进行的操作

*/
```

return ;

}

5

// 针对perm的第pos个位置，究竟使用0~n-1中的哪一个进行循环

for (int i = 0; i < n; i++) {

if (!used[i]) {

perm[pos] = i;

// i已经被使用了，所以把标志设置为true

6

used[i] = true;

permutation1(pos + 1, n);

// 返回之后把标志复位

used[i] = false;

}

}

7

return ;

}

#include <algorithm>

8

```
// 即使有重复的元素也会生成所有的排列
// next_permutation是按照字典序来生成下一个排列的
int perm2[MAX_N];
void permutation2(int n) {
```

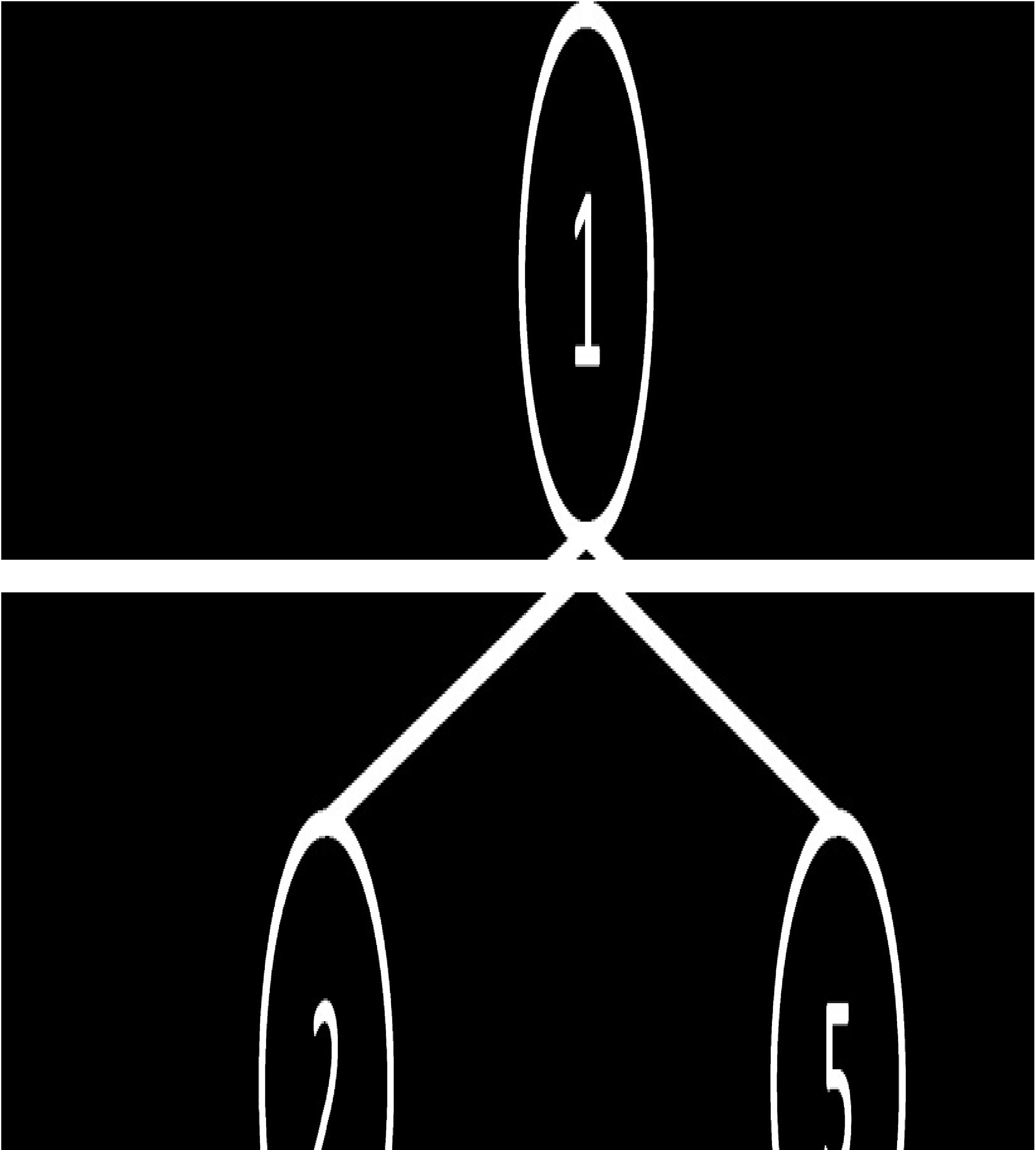
9

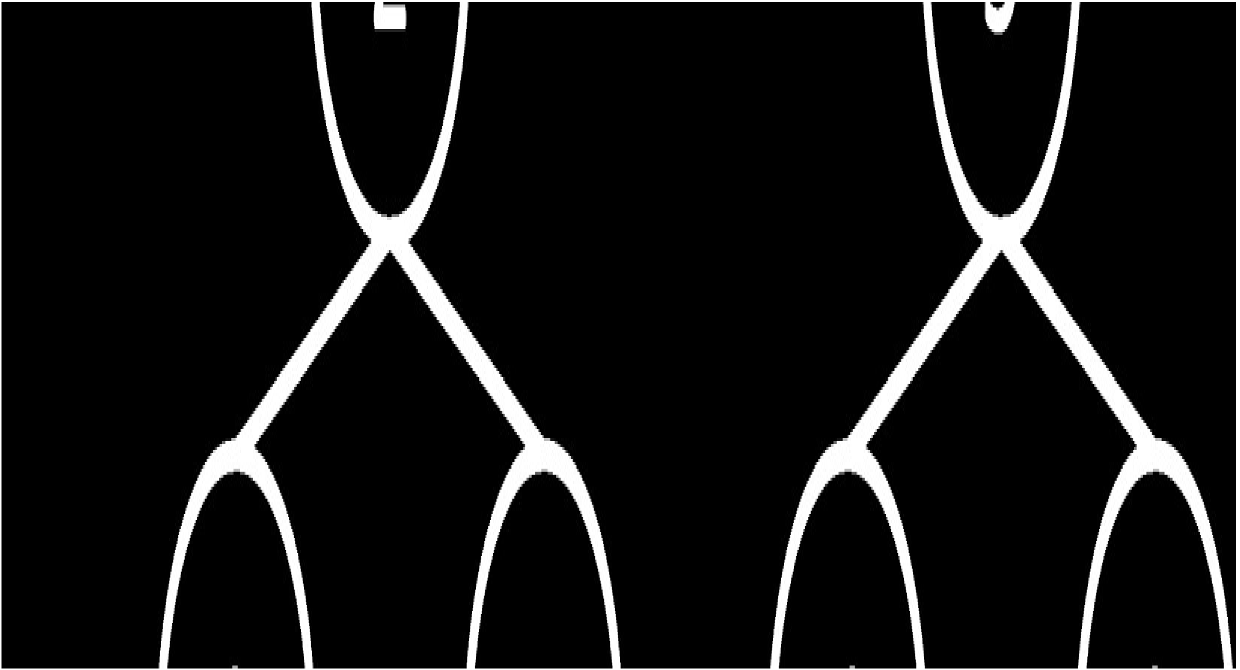
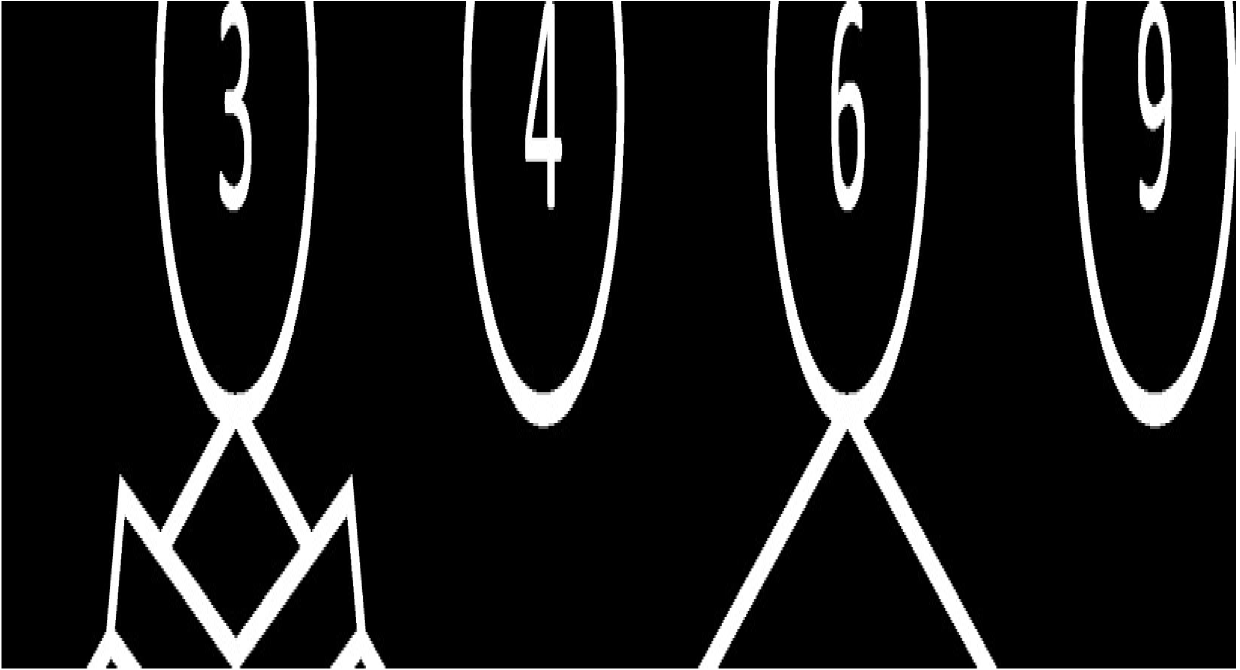
```
for (int i = 0; i < n; i++) {
perm2[i] = i;
}
do {
/*
* 这里编写需要对perm2进行的操作
```

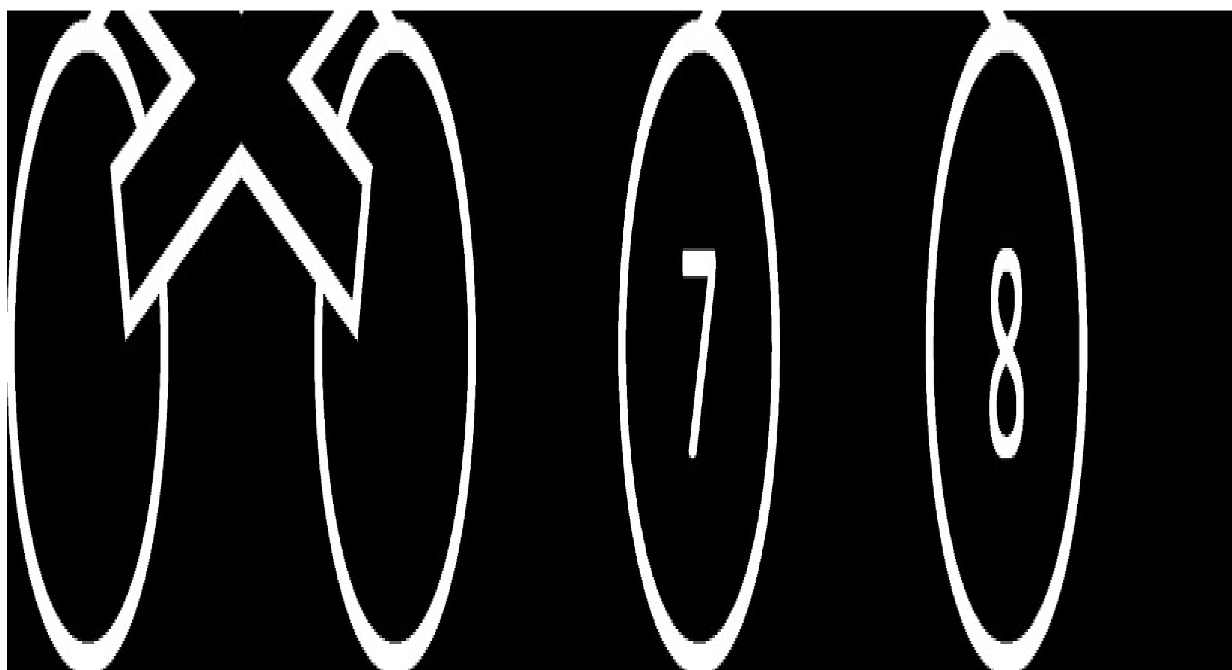
10

```
*/
} while (next_permutation(perm2, perm2 + n));
// 所有的排列都生成后，next_permutation会返回false
return ;
}
```

11


1
2
5



3
4
6
9



2.1.7 剪枝

顾名思义，穷竭搜索会把所有可能的解都检查一遍，当解空间非常大时，复杂度也会相应变大。

比如 n 个元素进行排列时状态数总共有 $n!$个，复杂度也就成了 $O(n!)$。这样的话，即使 $n=15$计算也

很难较早终止。这里简单介绍一下此类情形要如何进行优化。

深度优先搜索时，有时早已很明确地知道从当前状态无论如何转移都不会存在解。这种情况下，

不再继续搜索而是直接跳过，这一方法被称作剪枝。

我们回想一下深度优先搜索的例题“部分和问题”。这个问题中的限制条件如果变为$0 \le a_i \le 108$，

那么在递归中只要sum超过k了，此后无论选择哪些数都不可能让sum等于k，所以此后没有必要

继续搜索。

剪枝的情况

关于更多更高级的搜索手段，我们会在4.5节进行详细介绍。

专栏 栈内存和堆内存

调用函数时，主调的函数所拥有的局部变量等信息需要存储在特定的内存区域。这个区域被

称作栈内存区。另一方面，利用 new 或者 malloc 进行分配的内存区域被称为堆内存。

栈内存在程序启动时被统一分配，此后不能再扩大。由于这一区域有上限，所以函数的递归

深度也有上限。虽然与函数中定义的局部变量的数目有关，不过一般情况下 C 和 C++中进行

上万次的递归是可以的。在 Java 中，在执行程序时可以用参数指定栈的大小。不同的程序设

计竞赛所采用的设置各有不同，建议大家预先进行确认。GCJ 的话，程序是在自己的机器上

执行的，所以可以自行设置参数。

全局变量被保存在堆内存区。通常不推荐使用全局变量，但是在程序设计竞赛中，由于函数

通常不是那么多，并且常常会有多个函数访问同一个数组，因此利用全局变量就很方便。此

外，有时必须要申请巨大的数组，与放置在栈内存上相比，将其放置在堆内存上可以减少栈

溢出的危险。同时，通常只需定义满足最大需要的数列大小，但如果再额外定义大一些，能

很好地避免粗心导致的诸如忘记保留字符串末尾的'\0'的空间之类的漏洞。



1

2.7 一起来挑战GCJ的题目（1）

2

让我们一起运用迄今所介绍的技巧，实际挑战一下GCJ的题目吧。

3

2.7.1 Minimum Scalar Product

4

Minimum Scalar Product（2008 Round1A A）

有两个向量 $v_1=(x_1, x_2, ..., x_n)$和 $v_2=(y_1, y_2, …, y_n)$，允许任意交换 v_1 和 v_2 各自的分量的顺序。请

计算 v_1 和 v_2 的内积 $x_1y_1+…+x_ny_n$ 的最小值。

5

限制条件

Small

□ $1 \le n \le 8$

□ □$1000 \le x_i , y_i \le 1000$

6

Large

□ $100 \le n \le 800$

□ □$100000 \le x_i , y_i \le 100000$

7

样例 1

输入

8

n = 3

v = (1, 3, -5)

1

v = (-2, 4, 1)

2

输出

9

令v = (-5, 1, 3), v = (4, 1, -2)就可以得到最小值v × v = -25

1

2

1

2

样例 2

10

输入

n = 5

v = (1, 2, 3, 4, 5)

1

v = (1, 0, 1, 0, 1)

2

11

输出

令v = (1, 2, 3, 4, 5), v = (1, 1, 1, 0, 0)就可以得到最小值6

1

2

v 1和 v 2各自的分量的顺序都可以任意交换，因此可以先把 v 1的顺序固定下来只交换 v 2的顺序。为了

方便分析先将 v 1按升序排好序。接下来枚举 v 2的分量所有的排列顺序，一共有 n !种排列，还需要

对每种排列计算内积，总的复杂度是 $O(n!\times n)$。这个算法在Small的情况，因为 $n\leq 8$所以没有问题，

但在Large的情况时就远远不够了。

在此隐约会觉得把 v_2按降序或者升序排序的话，所得的内积是最小的。事实上，如果将 v_2按降序

排序的话，所给的两个样例都能够得到最小值。这个设想的确是对的， v_1和 v_2的内积在将 v_1按升

序，将 v_2按降序排序时取得最小值。下面我们来证明这一设想。首先考虑 $n=2$的情况。

考虑 $v_1=(x_1, x_2), v_2=(y_1, y_2)$，假设 v_1已经按升序排好序了，即有 $x_1 \le x_2$，比较 $x_1 \times y_1 + x_2 \times y_2$和 $x_2 \times y_1$

$+ x_1 \times y_2$的大小关系。

$x_1 \times y_1 + x_2 \times y_2 \square x_2 \times y_1 \square x_1 \times y_2 = x_1 (y_1 \square y_2) + x_2 (y_2 \square y_1) = (x_1 \square x_2)(y_1 \square y_2)$ 由此可知， $y_1 \ge y_2 \square x_1 \times y_1 + x_2 \times y_2 \le x_2 \times y_1 + x_1 \times y_2$。因此 $n=2$时结论成立。

接下来考虑 n 大于2的情况。如果 v_2不是按降序排序的，那么存在 $i < j$ 使得 $y_i < y_j$ ，根据对 $n=2$的情

况的分析可以知道，交换 y_i 和 y_j 后就得到了更小的内积。因此，当将 v_2按降序排序时，所得的内积

最小。

数组排序的复杂度为 $O(n \log n)$，所以发现这一结论后，只要做两次排序就可以简单高效地计算答

案了。那么，要怎样才能去想到这个设想呢?

首先就是靠直觉。比较容易想到的姑且先排序试试看。

第二个就是样例。原本的问题描述中并没有说明 v_1和 v_2在什么情况下取得最小值。但是，因为样

例的规模比较小，所以可以手工验算。因而可以找到使得内积最小的 v_1和 v_2。通过观察对应的 v_1

和 v_2，想要得到刚才的结论也并不是那么难。

第三个就是像证明中的那样，从像 n =2这样小规模的情况出发，推广到一般的情况从而证明结

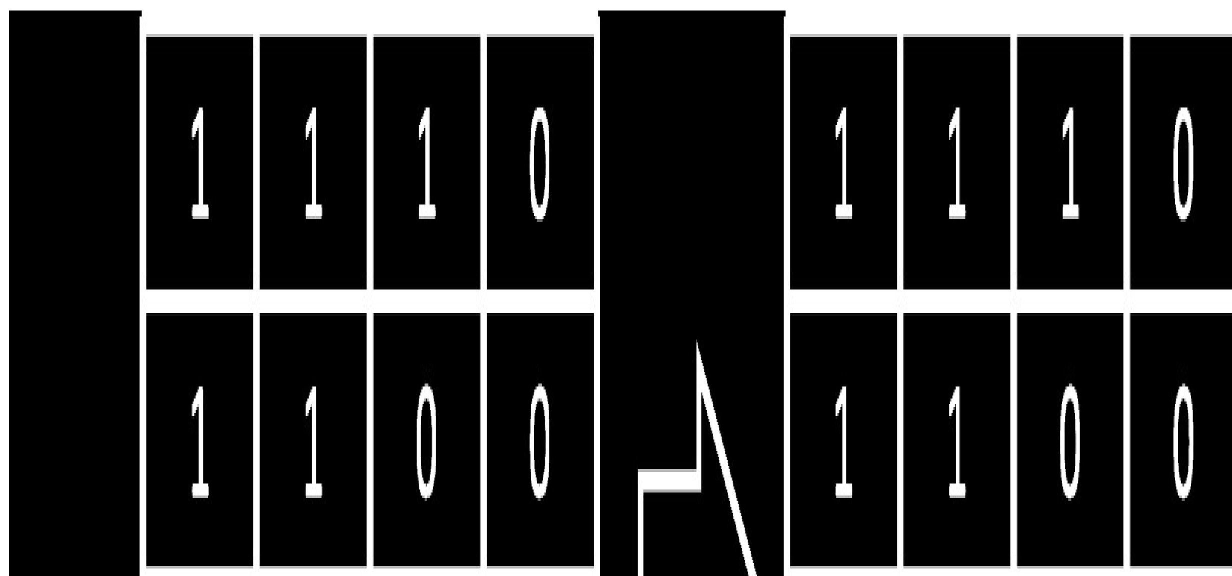
论。在这个问题中，很容易证明对 n =2的情况结论成立，接着同样可以证明对一般的情况结论

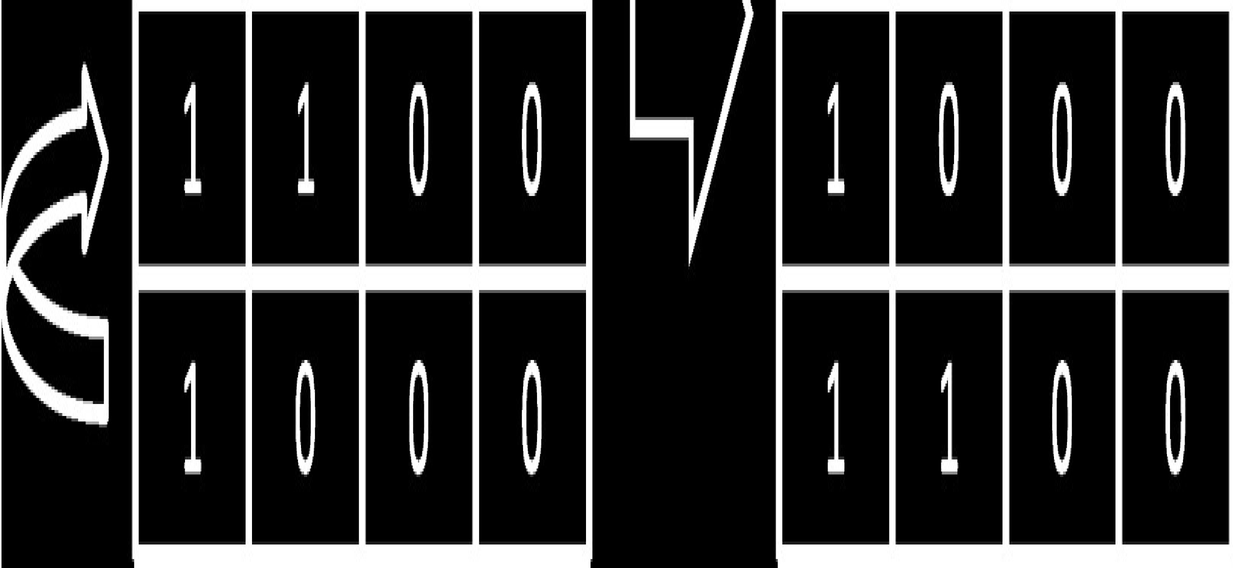
都成立。

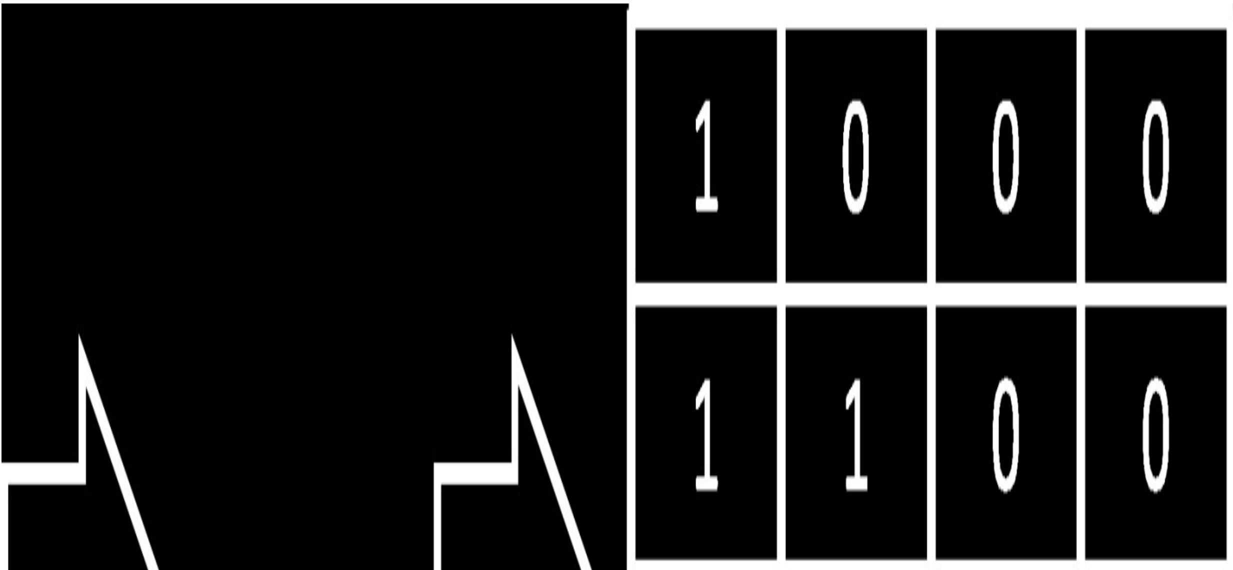
最后需要注意的是，即使推导出了正确的算法，如果程序的实现中有漏洞的话也是徒劳。在这道

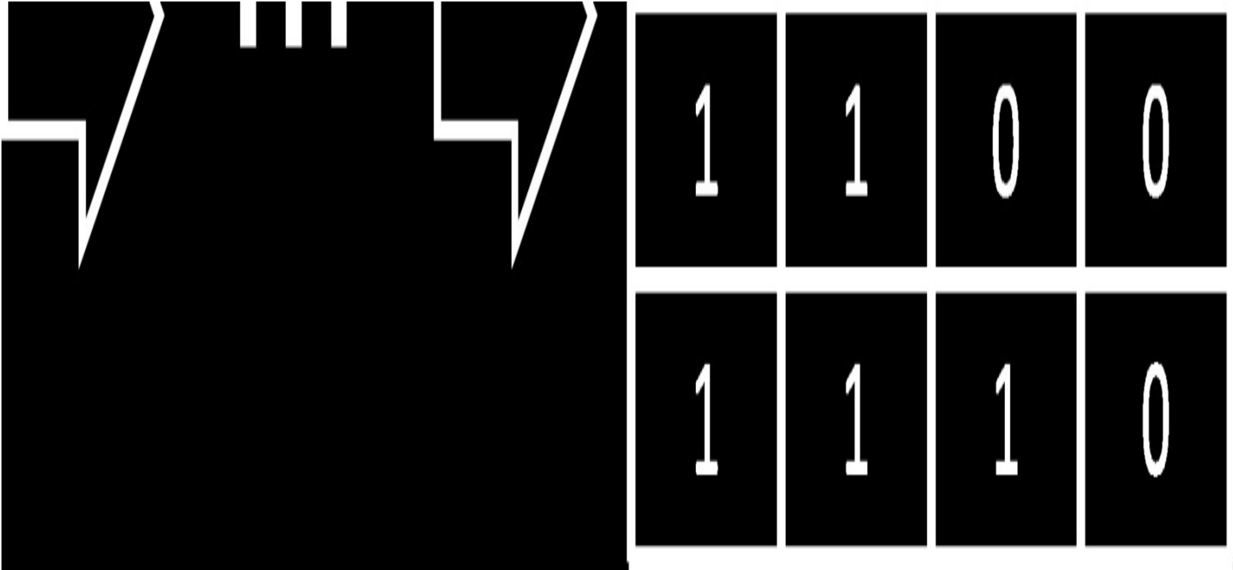
题中， v_1的分量和 v_2的分量的乘积可能会导致32位整数溢出。即使认为找到了正解，不到最后提

交结果正确的那一刻，都不能掉以轻心呢。




1 1 0 0
1 0 0 0
1 0 0 0
1 1 0 0


1 0 0 0
1 1 0 0






```
typedef long long ll;
```

1

```
// 输入
int n;
int v1[MAX_N], v2[MAX_N];
```

2

```
void solve() {
sort(v1, v1 + n);
```

```
sort(v2, v2 + n);

ll ans = 0;

for (int i = 0; i < n; i++) ans += (ll)v1[i] * v2[n - i - 1];
```
3

```
printf("%lld\n", ans);

}
```

2.7.2 Crazy Rows

4

Crazy Rows （2009 Round2 A）

5

给定一个由 0 和 1 组成的矩阵。只允许交换相邻的两行（第 *i* 行和第 *i* +1 行），要把矩阵化成

下三角矩阵（主对角线上方的元素都是 0），最少需要交换几次?输入的矩阵保证总能化成下

三角矩阵。

6

7

行交换

变换后

限制条件

Small

 $1 \le N \le 8$

8

Large

- $1 \leq N \leq 40$

9

样例 1

输入

N = 2

10

矩阵

10

11

11

输出

0(输入已经是下三角矩阵)

样例 2

输入

N = 3

矩阵

001

100

010

输出

2(交换第1行和第2行，再交换第2行和第3行，得到下三角矩阵)

样例 3

输入

N = 4

矩阵

1110

1100

1100

1000

输出

4

最先想到的是尝试所有 N !种交换方案。但在Large中，由于最大的 N =40，这当然是行不通的。

暂且先考虑一下最后应该把哪一行交换到第1行。最后的第1行应该具有00...0或是10...0的形式。

可以交换到第1行的行当然也可以交换到第2及之后的行，当有多个满足条件的行时，选择离第1

行近的行对应的最终费用要小。大家肯定都已注意到了这一点吧。有兴趣的读者不妨自己证明

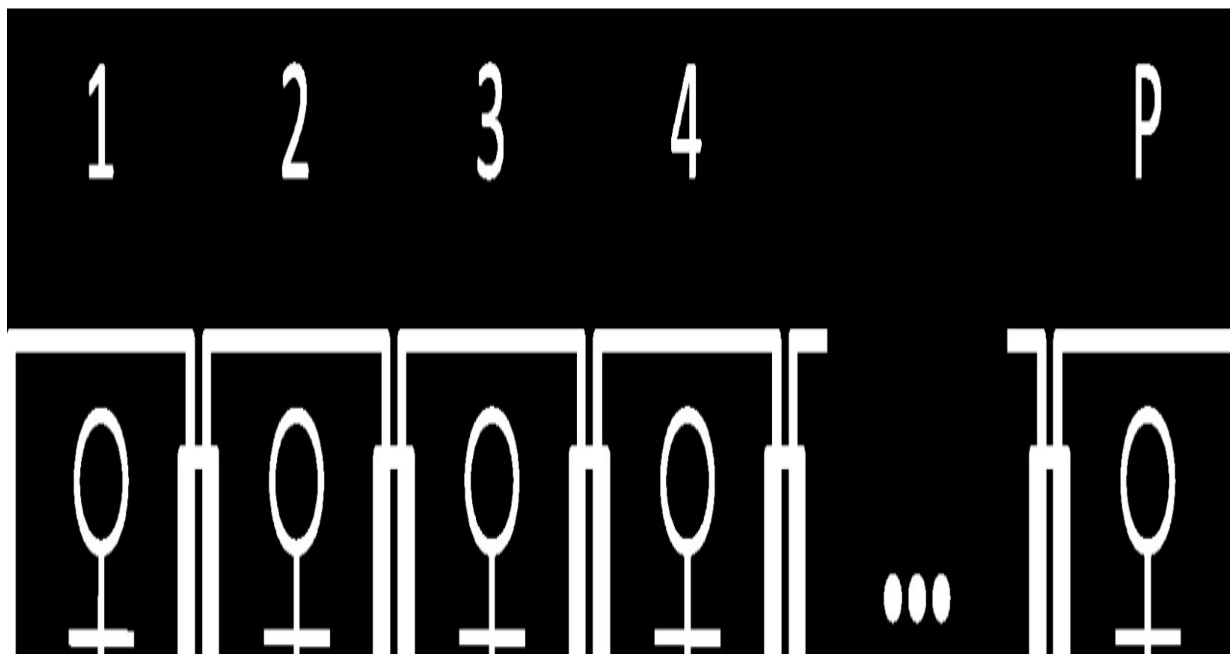
一下。

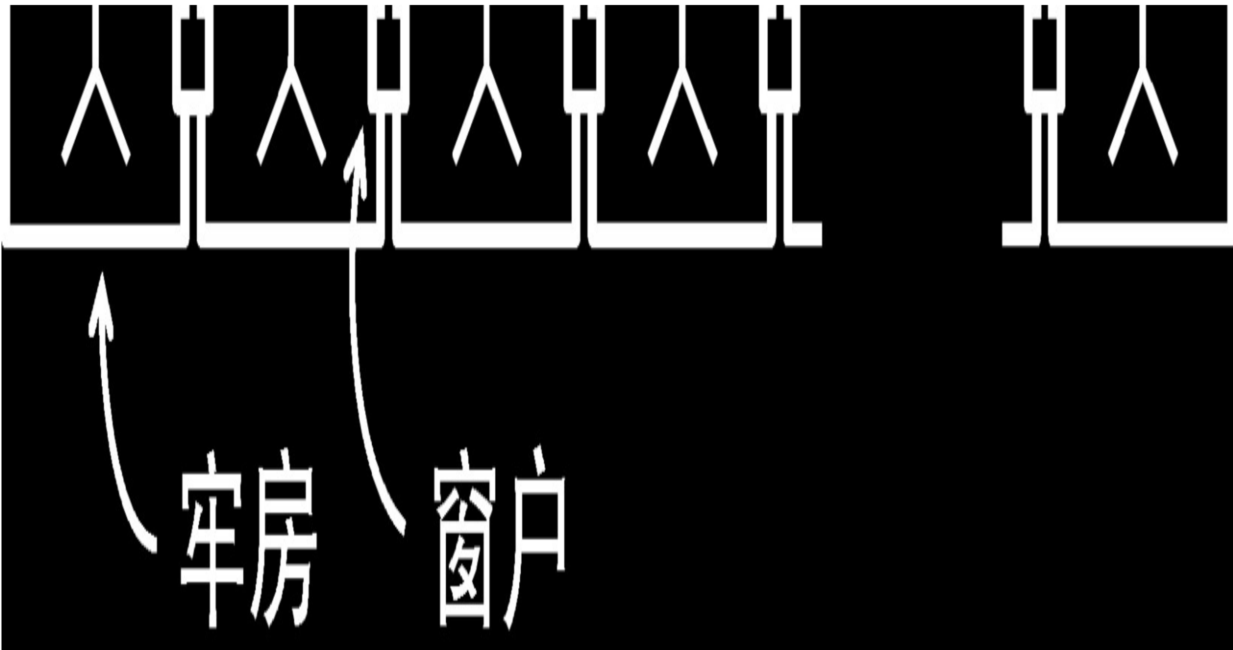
确定第1行之后，就没有必要再移动它了，于是对于之后的行就可以以同样的思路处理。

在这道题中，每行的0和1的位置并不重要，只要知道每行最后一个1所在的位置就足够了。如果

先将这些位置预先计算好，那么就能降低行交换时的复杂度。直接按矩阵的形式处理的复杂度是

$O(N^3)$，而预先计算后再处理的复杂度降为 $O(N^2)$。





// 输入

1

```
int N;
int M[MAX_N][MAX_N]; // 矩阵
int a[MAX_N]; // a[i]表示第i行最后出现的1的位置——1~n-1
```

2

```
void solve() {
int res = 0;
for (int i = 0; i < N; i++) {
a[i] = -1; // 如果第i行不含1的话，就当作-1
for (int j = 0; j < N; j++) {
```

3

```
if (M[i][j] == 1) a[i] = j;
}
}
for (int i = 0; i < N; i++) {
int pos = -1; // 要移动到第i行的行
for (int j = i; j < N; j++) {
```

4

```
if (a[j] <= i) {
pos = j;
break;
}
}
```

5

```
// 完成交换
for (int j = pos; j > i; j--) {
swap(a[j], a[j - 1]);
res++;
```

6

```
}
```

```
}
printf("%d\n", res);
}
```

7

2.7.3 Bribe the Prisoners

8

Bribe the Prisoners （2009 Round 1C C）

如下图所示，一个监狱里有 P 个并排着的牢房。从左至右依次编号为 1, 2, …, P 。最初所有的牢

房里都住着一个囚犯。相邻的两个牢房之间有一个窗户，可以通过它与相邻牢房里的囚犯对话。

9

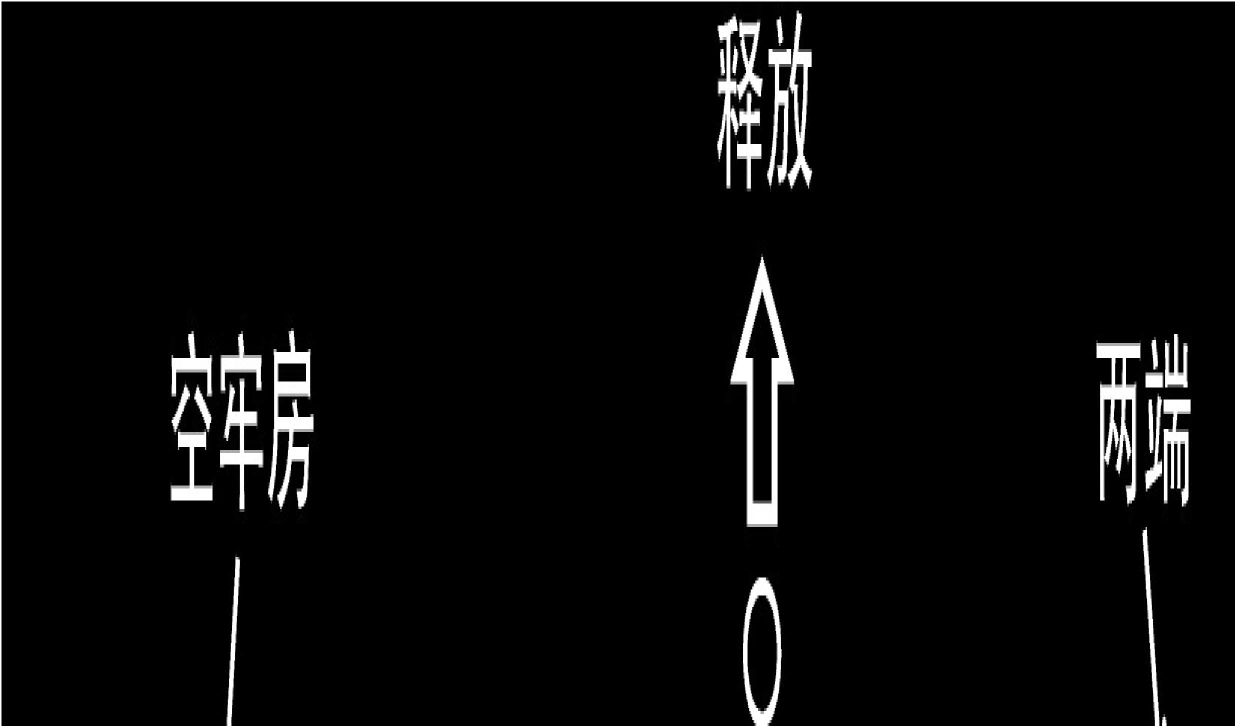
10

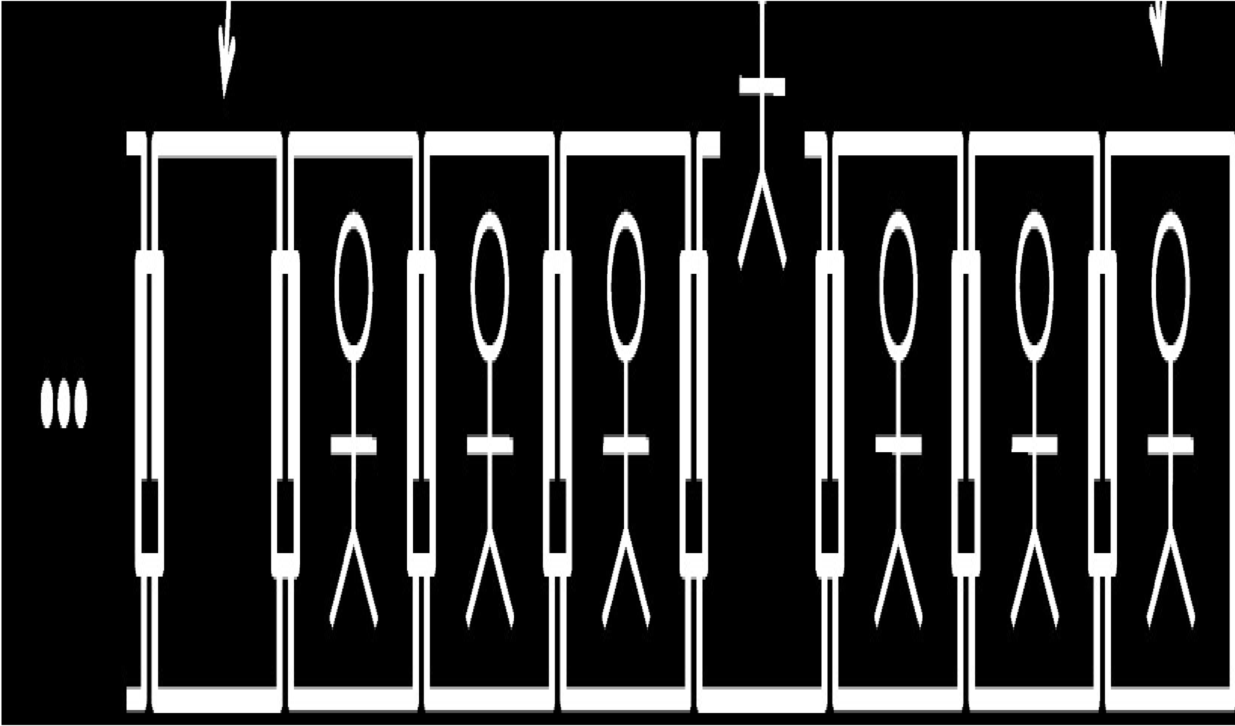
监狱的情况

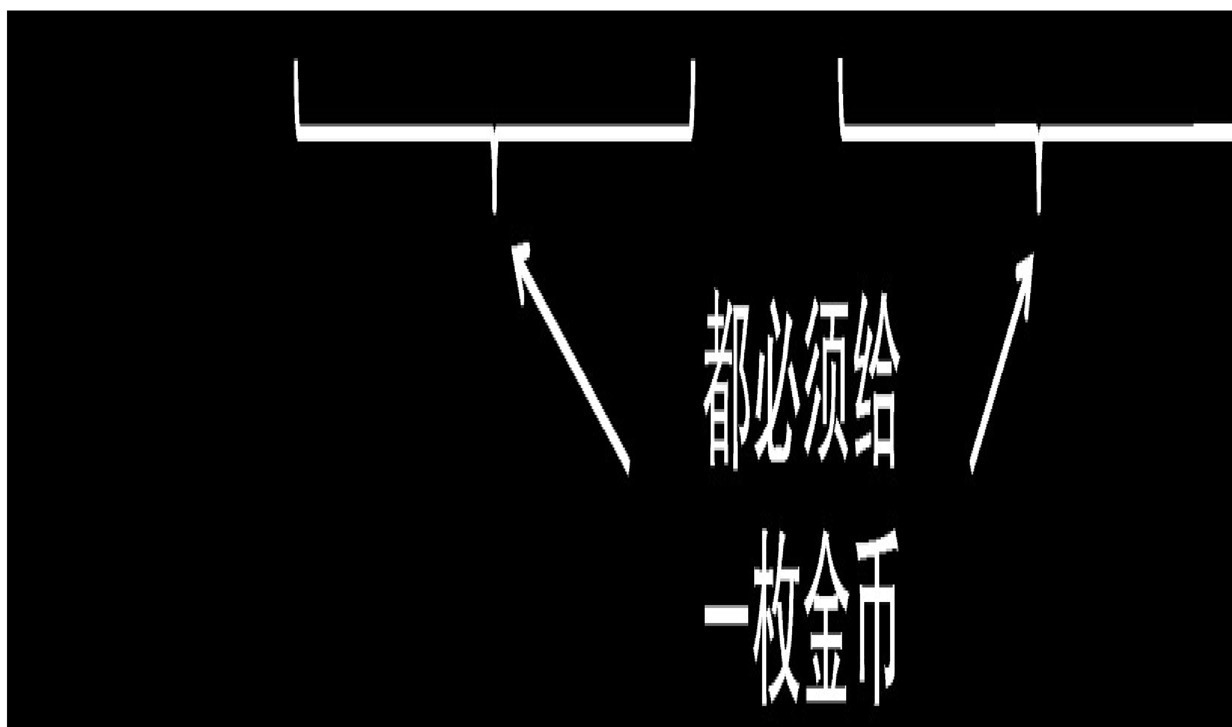
现在要释放一些囚犯。如果释放某个牢房里的囚犯，其相邻的牢房里的囚犯就会知道，因而

11

发怒暴动。所以，释放某个牢房里的囚犯同时，必须要贿赂两旁相邻牢房里的囚犯一枚金币。


释放
空牢房
两端







另外，为了防止释放的消息在相邻牢房间传开，不仅两旁直接相邻的牢房，所有可能听到消

息的囚犯，即直到空牢房为止或直到监狱两端为止,此间的所有囚犯都必须给一枚金币。

释放后给金币的例子

现在要释放 $a_1, a_2, \ldots, a_Q$ 号牢房里的 Q 名囚犯，释放的顺序还没确定。如果选择所需金币数

量尽量少的顺序释放，最少需要多少枚金币？

限制条件

- $1 \le N \le 100$

- $Q \le P$

Small

- $1 \le P \le 100$

- $1 \le Q \le 5$

Large

- $1 \le P \le 10000$

- $1 \le Q \le 100$

样例 1

输入

P = 8, Q = 1, A = {3}

输出

7（必须要给剩下的7个人一枚金币）

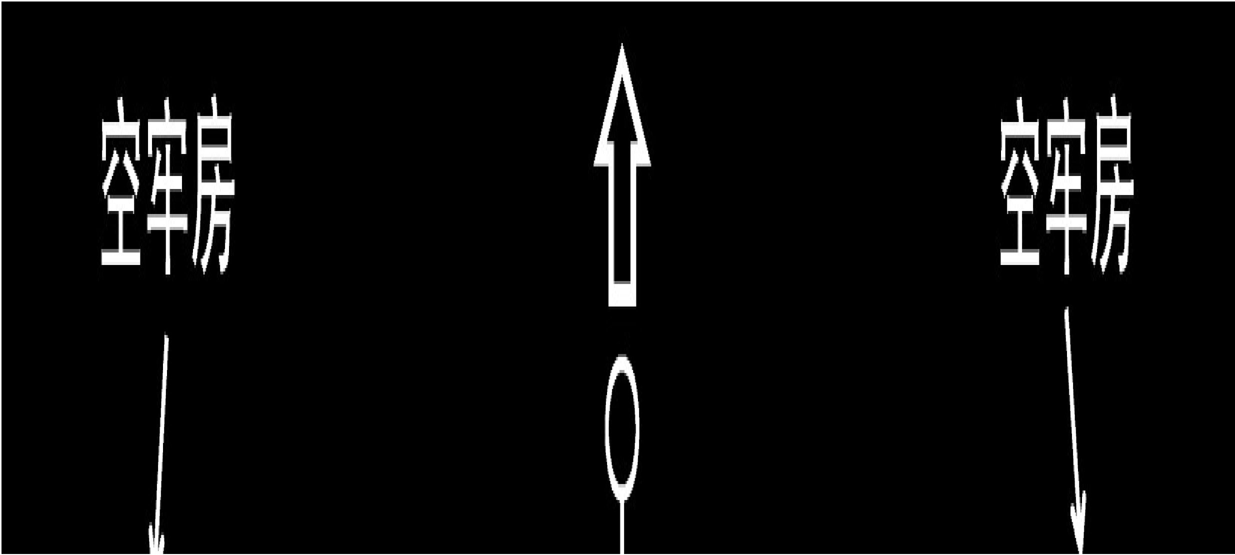
样例 2

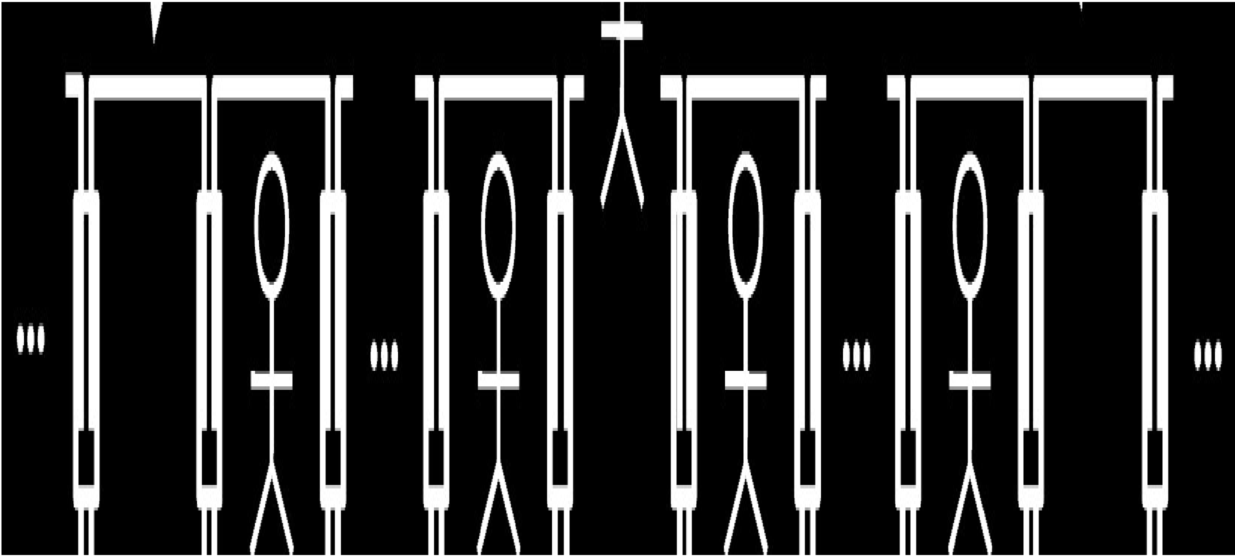
输入

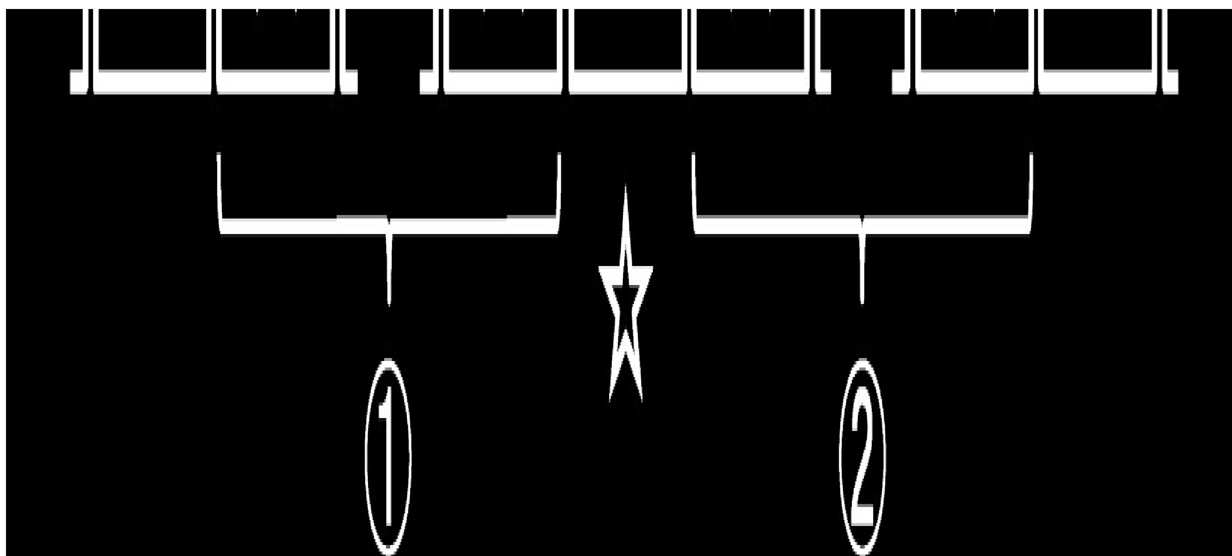
P = 20, Q = 3, A = {3, 6, 14}

输出

35(按照牢房14、牢房6、牢房3的顺序释放，则需要19 + 12 + 4即35枚金币，是最少的)


空牢房
空牢房





这道题的关键是，释放了某个囚犯之后，就把连续的牢房分成了两段，此后这两段就相互独立了。 **1**

2

释放后分成两部分的情况

3

释放上图中☆里的囚犯时

 此时所需的金币数量

 释放左侧部分（①）所需的金币总数

4

 释放右侧部分（②）所需的金币总数

这三者的总和就是所需的金币总数。只要不断递归地枚举最初释放的囚犯并计算对应的金币总

数，就能求出答案了。

5

这里，递归计算过程中作为计算对象的连续部分，其两端是空牢房或监狱两端。因此，作为计算

对象的连续部分一共有 $O(Q^2)$个。所以，利用动态规划按顺序计算，就能够在 $O(Q^3)$时间内求解。

6

```
// 输入

int P, Q, A[MAX_Q + 2]; // A中保存输入数据，下标从1开始

// dp[i][j] := 释放(i, j)所需的金币①

int dp[MAX_Q + 1][MAX_Q + 2];
```

7

```
void solve() {

// 为了处理方便，将两端加入A中②

A[0] = 0;

A[Q + 1] = P + 1;
```

8

```
// 初始化

for (int q = 0; q <= Q; q++) {

dp[q][q + 1] = 0;

}
```

9

// 从短的区间开始填充dp

for (int w = 2; w <= Q + 1; w++) {

for (int i = 0; i + w <= Q + 1; i++) {

// 计算dp[i][j]

int j = i + w, t = INT_MAX;

10

———————————

① dp[i][j]表示的是，将从A[i]号囚犯到A[j]号囚犯（不含两端的囚犯）的连续部分里的所有囚犯都释放时,所需的最少金

币总数。

11

② 为了更方便地处理两端的情况，我们把左端当成0号囚犯，右端当成Q+1号囚犯。这样，dp[0][Q+1]就是答案。



// 枚举最初释放的囚犯，计算最小的费用

for (int k = i + 1; k < j; k++) {

t = min(t, dp[i][k] + dp[k][j]);

```
}
// 最初的释放还需要与所释放囚犯无关的A[j]-A[i]-2枚金币
dp[i][j] = t + A[j] - A[i] - 2;
}
}
printf("%d\n", dp[0][Q + 1]);
}
```

2.7.4 Millionaire

Millionaire （2008 APAC local onsites C）

你被邀请到某个电视节目中去玩下面这个游戏。一开始你有 *x* 元钱，接着进 *M* 轮赌博。每

一轮，可以将所持的任意一部分钱作为赌注。赌注不光可以是整数，也可以是小数。一分钱

不押或全押都没有关系。每一轮都有 *P* 的概率可以赢，赢了赌注就会翻倍，输了赌注就没了。

如果你最后持有 1000000 元以上的钱的话，就可以把这些钱带回家。请计算当你采取最优策

略时，获得 1000000 元以上的钱并带回家的概率。

限制条件

 $0 \le P \le 1.0$

 $1 \le X \le 1000000$

Small

□ $1 \le M \le 5$

Large

□ $1 \le M \le 15$

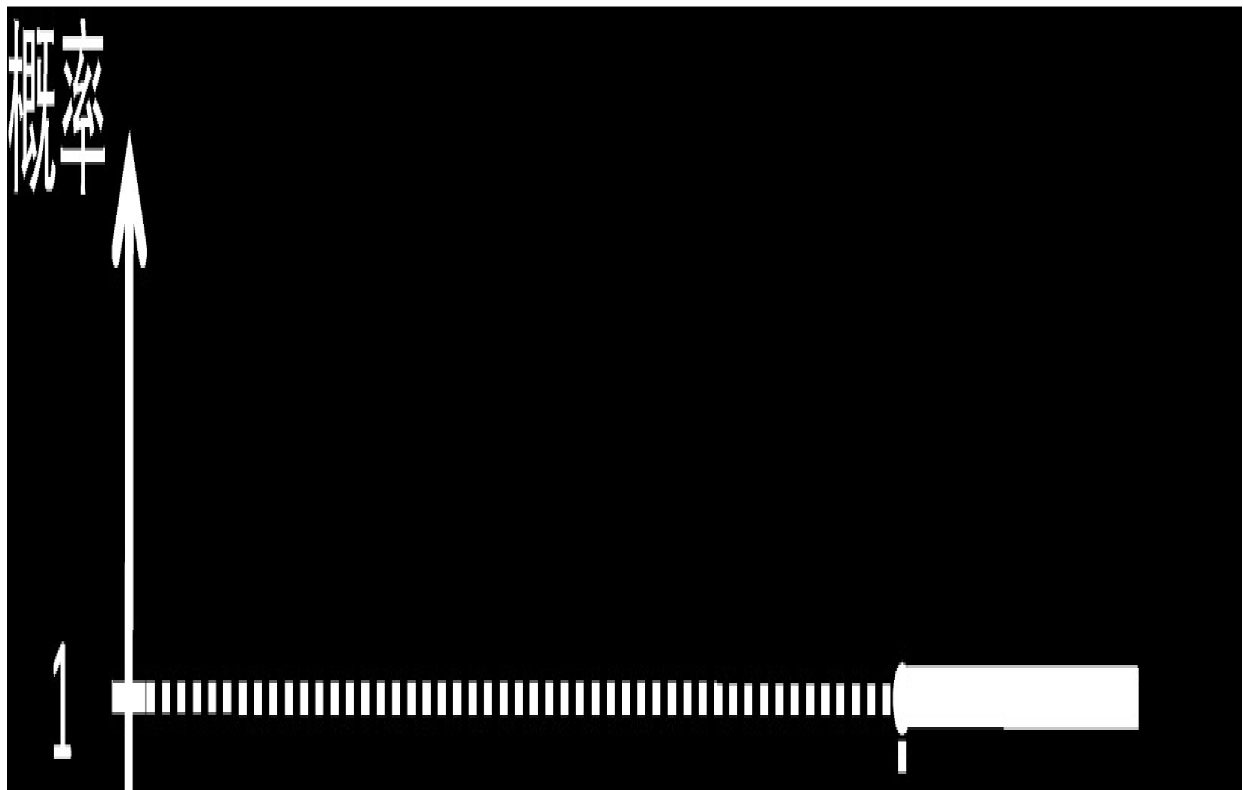
样例 1

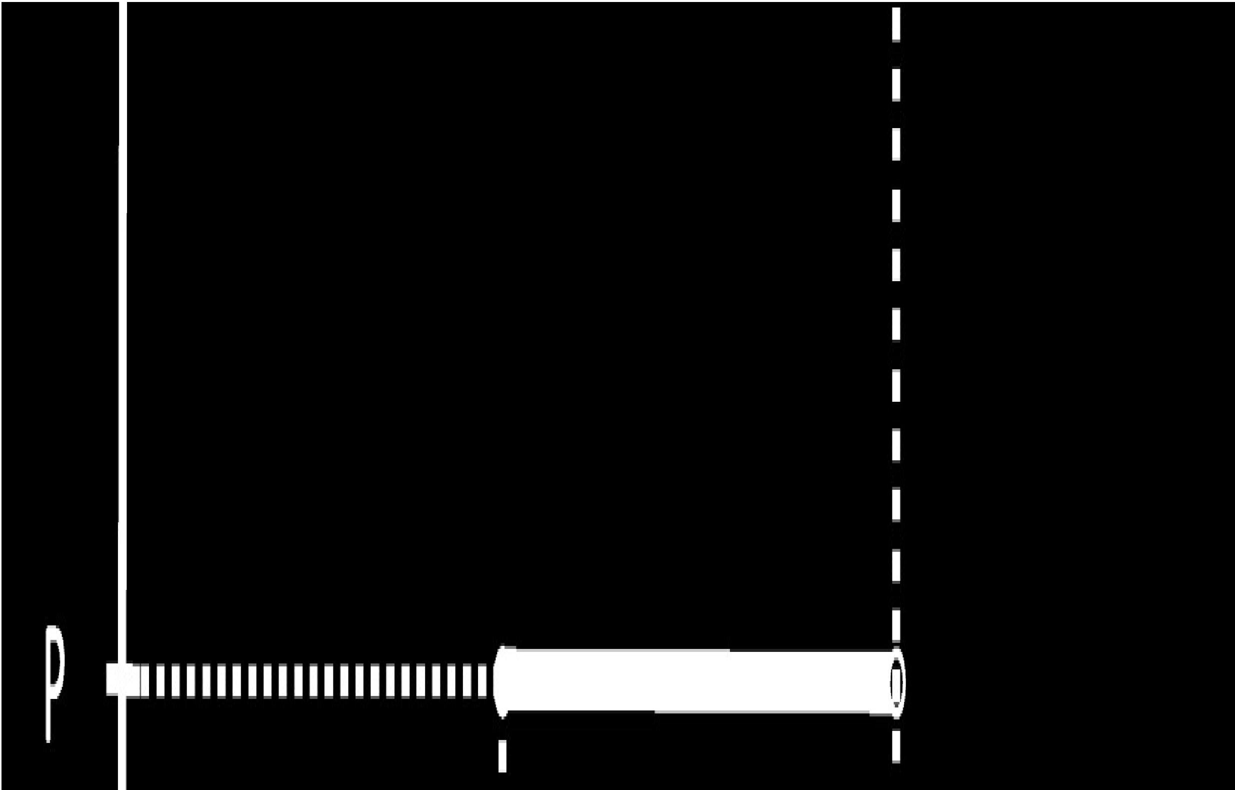
输入

M = 1, P = 0.5, X = 500000

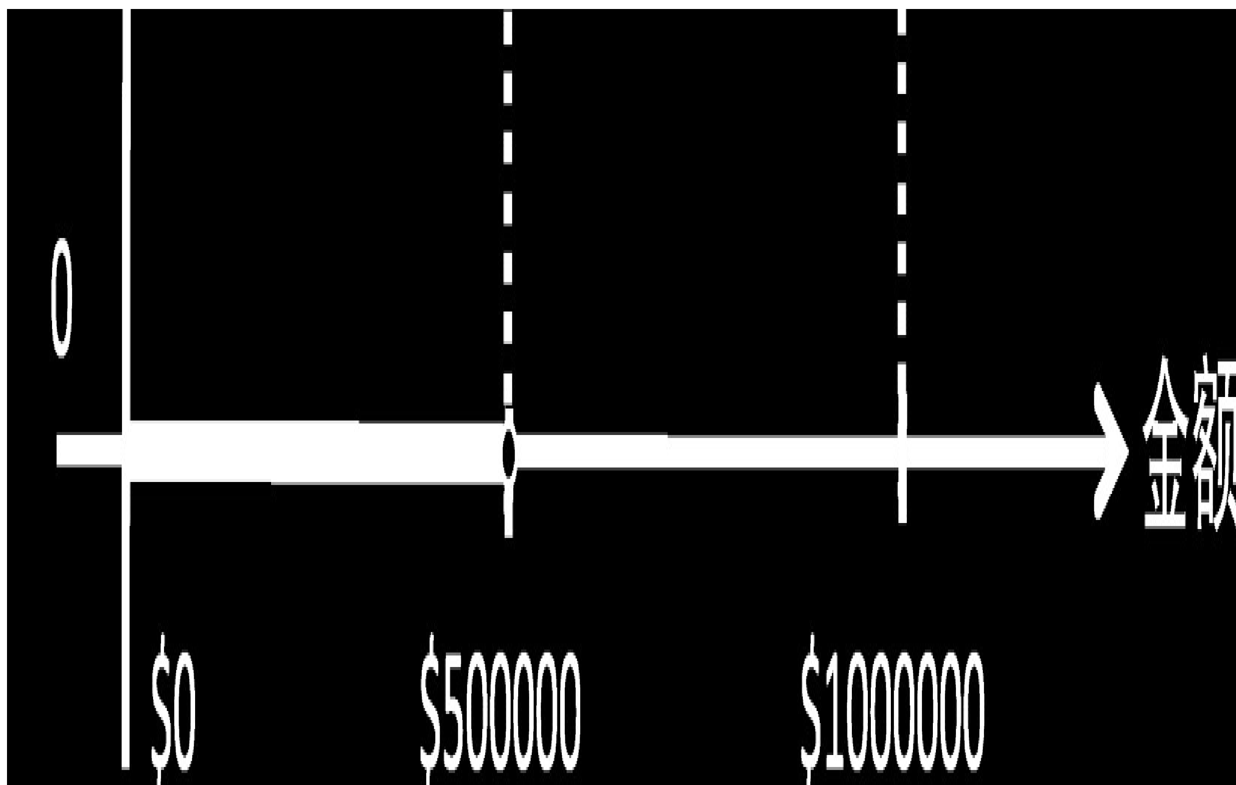
输出

0.500000(一开始便全押)




P





样例 2

1

输入

M = 3, P = 0.75, X = 600000

2

输出

0.843750

1. 该问题的难点

3

“连续性”是这个问题的一大特点。每一轮可押的赌注不一定非是整数，因而有无限种可能，所

以完全无法穷竭搜索。

4

2. 化连续为离散

但是，认真思考一下就会发现，我们只需要检查 “无限种可能”中的“有限种可能”就足够了，

因而可以设计对应的算法。首先来思考一下最后一轮会出现哪些情况。

5

- 如果你持有1000000元以上的钱，就没有再赌的必要了。有1的概率可以带钱回家。

- 如果你持有500000元以上的钱，不妨全押了。有 P 的概率可以带钱回家。虽然不全押也是可以

的，不过因为要求要达到1000000元，所以不论怎样都是这轮赢了就能带钱回家，输了就不能

6

带钱回家。

- 如果你持有不到500000元的钱，那已经无可奈何了。有0的概率可以带钱回家。

7

8

9

最后一轮带钱回家的概率

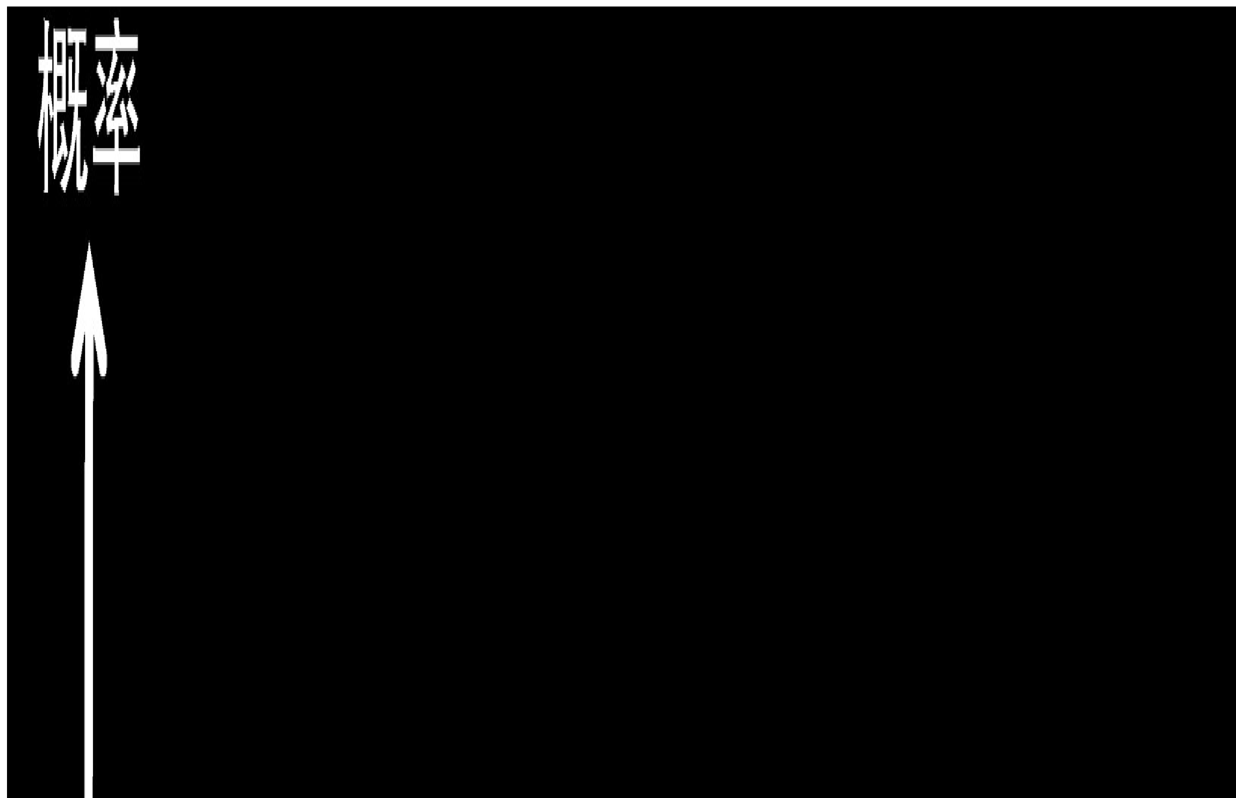
虽然赌注是连续的（有无限种可能），但实际的概率是像上面这样的阶梯函数。因此，只需考虑

10

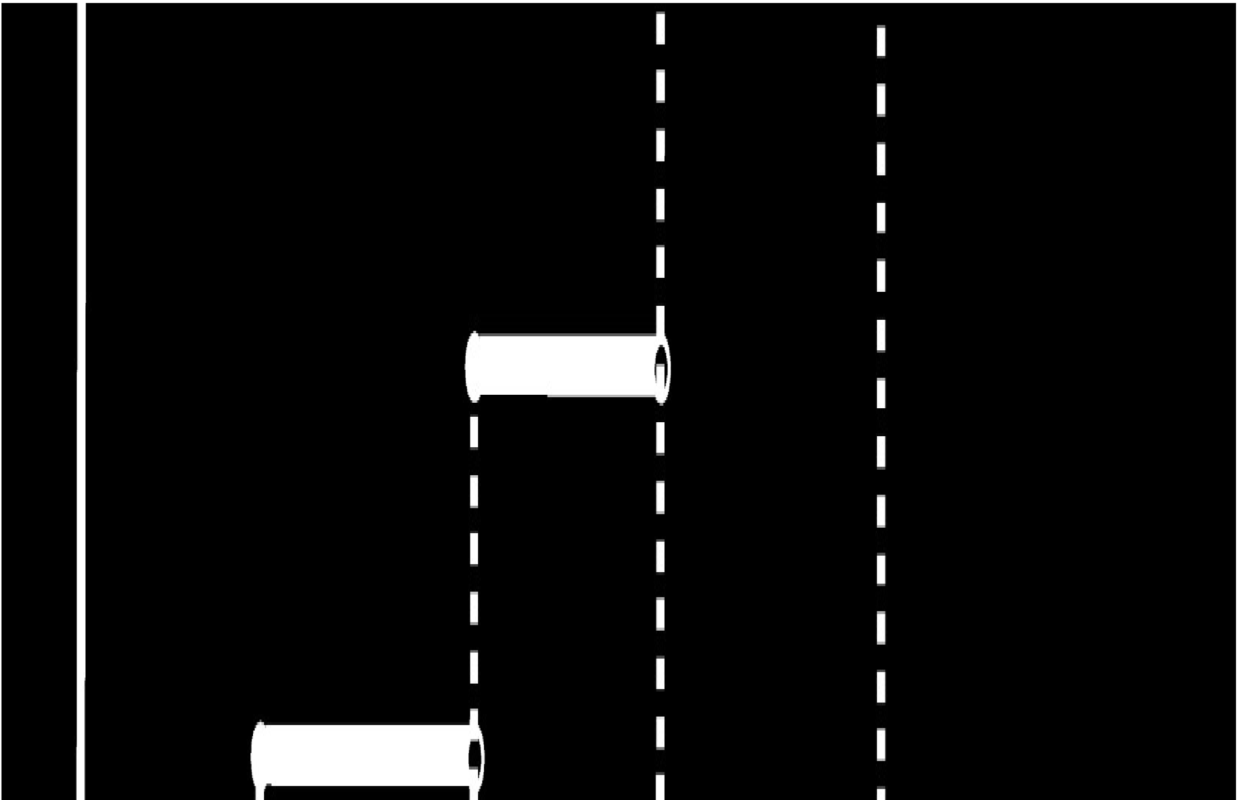
处于这三个范围中的哪一个就可以了。

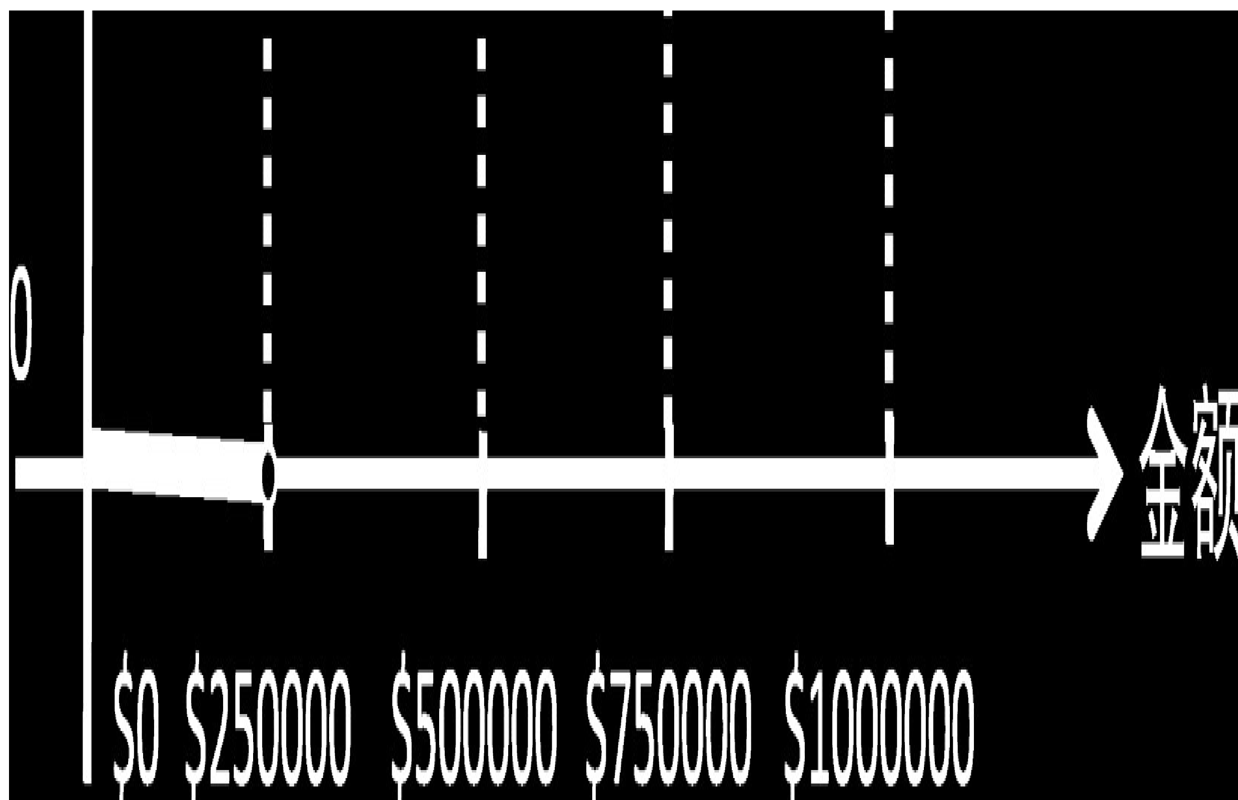
那么，最后两轮时又如何呢？同样地，这次也分为5种情况。

11









最后两轮带钱回家的概率

某个范围中，即使所持的钱数不同，最后可以带钱回家的概率也是完全一样的。而且同样地， M

轮时只要考虑2 M +1种情况就足够了。这样，就可以设计出穷竭搜索的算法了。

3. 动态规划

接下来，将穷竭搜索改成动态规划，就能够求解Large了。

```
// 输入

int M, X;

double P;
```

```
double dp[2][(1 << MAX_M) + 1];

void solve() {

int n = 1 << M;

double *prv = dp[0], *nxt = dp[1];

memset(prv, 0, sizeof(double) * (n + 1));

prv[n] = 1.0;

for (int r = 0; r < M; r++) {

for (int i = 0; i <= n; i++) {

int jub = min(i, n - i);

double t = 0.0;

for (int j = 0; j <= jub; j++) {

t = max(t, P * prv[i + j] + (1 - P) * prv[i - j]);

}

nxt[i] = t;

}

swap(prv, nxt);

}

int i = (ll)X * n / 1000000;

printf("%.6f\n", prv[i]);

}
```

练 习 题

1

2.1 最基础的“穷竭搜索”

POJ 1631: Bridging signals

□

POJ 3666: Making the Grade

深度优先搜索

POJ 2392: Space Elevator

POJ 1979: Red and Black

POJ 2184: Cow Exhibition

2

AOJ 0118: Property Distribution

2.4 加工并存储数据的数据结构

AOJ 0033: Ball

POJ 3009: Curling 2.0

□ 优先队列

□ 广度优先搜索

POJ 3614: Sunscreen

POJ 2010: Moo University - Financial Aid

3

AOJ 0558: Cheese

POJ 3669: Meteor Shower

 并查集

AOJ 0121: Seven Puzzle

POJ 2236: Wireless Network

 穷竭搜索

POJ 1703: Find them, Catch them

AOJ 2170: Marked Ancestor

4

POJ 2718: Smallest Difference

POJ 3187: Backward Digit Sums

2.5 它们其实都是“图”

POJ 3050: Hopscotch

 最短路

AOJ 0525: Osenbei

AOJ 0189: Convenient Location

2.2 一往直前！贪心法

5

POJ 2139: Six Degrees of Cowvin Bacon

□ 区间

POJ 3259: Wormholes

POJ 2376: Cleaning Shifts

POJ 3268: Silver Cow Party

POJ 1328: Radar Installation

AOJ 2249: Road Construction

AOJ 2200: Mr. Rito Post Office

6

POJ 3190: Stall Reservations

□ 其他

□ 最小生成树

POJ 2393: Yogurt factory

POJ 1258: Agri-Net

POJ 1017: Packets

POJ 2377: Bad Cowtractors

AOJ 2224: Save your cat

7

POJ 3040: Allowance

POJ 1862: Stripies

POJ 2395: Out of Hay

POJ 3262: Protecting the Flowers

2.6 数学问题的解题窍门

2.3 记录结果再利用的“动态规划”

□ 辗转相除法

8

□ 基础的动态规划算法

AOJ 0005: GCD and LCM

POJ 3176: Cow Bowling

POJ 2429: GCD & LCM Inverse

POJ 2229: Sumsets

POJ 1930: Dead Fraction

POJ 2385: Apple Catching

□ 素数

9

POJ 3616: Milking Time

AOJ 0009: Prime Number

POJ 3280: Cheapest Palindrome

POJ 3126: Prime Path

□ 优化递推关系式

POJ 3421: X-factor Chains

POJ 1742: Coins

POJ 3292: Semi-prime H-numbers

10

POJ 3046: Ant Counting

 快速幂运算

POJ 3181: Dollar Dayz

POJ 3641: Pseudoprime numbers

 需稍加思考的题目

POJ 1995: Raising Modulo Numbers

POJ 1065: Wooden Sticks

11

第 3 章

出类拔萃——中级篇



3.2 常用技巧精选（一）

在此我们介绍一些程序设计竞赛中的常用技巧。①

3.2.1 尺取法②

Subsequence （POJ No.3061）

给定长度为 n 的数列整数 $a_0, a_1, \ldots, a_{n-1}$ 以及整数 S 。求出总和不小于 S 的连续子序列的长度的

最小值。如果解不存在，则输出 0。

限制条件

- $10 < n < 105$
- $0 < a_i \le 104$
- $S < 108$

样例 1

输入

n = 10

S = 15

a = {5, 1, 3, 5, 10, 7, 4, 9, 2, 8}

输出

2 (5+10)

样例 2

输入

n = 5

---

① 本节所涉及的多种技巧，其应用范围都比这里所介绍的模型要广得多，注意蕴含在技巧内的算法思想。许多看似非常

复杂困难的问题，其问题关键都可以运用这里看似简单的技巧处理。——译者注

② 这里直接使用了日文原文的汉字，尺取法通常是指对数组保存一对下标（起点、终点），然后根据实际情况交替推进

两个端点直到得出答案的方法，这种操作很像是尺蠖（日文中称为尺取虫）爬行的方式故得名。——译者注

S = 11

1

a = {1, 2, 3, 4, 5}

输出

3 (3+4+5)

2

由于所有的元素都大于零，如果子序列$[s, t)$满足$a_s + \ldots + a_{t-1} \geq S$，那么对于任何的$t < t'$一定有$a_s + \ldots$

$+ a_{t'-1} \geq S$。此外对于区间$[s, t)$上的总和来说如果令

3

$\mathrm{sum}(i) = a_0 + a_1 + \ldots + a_{i-1}$

那么

$a_s + a_{s+1} + \ldots + a_{t-1} = \mathrm{sum}(t) - \mathrm{sum}(s)$ **4**

因此预先以 $O(n)$的时间计算好sum的话，就可以以 $O(1)$的时间计算区间上的总和。这样一来，子

序列的起点 s 确定以后，便可以用二分搜索快速地确定使序列和不小于 S 的结尾 t 的最小值。

5

```
// 输入

int n, S;

int a[MAX_N];

int sum[MAX_N + 1];
```

6

```
void solve() {

// 计算sum

for (int i = 0; i < n; i++) {

sum[i + 1] = sum[i] + a[i];
```

7

```
}

if (sum[n] < S) {

// 解不存在

printf("0\n");
```

8

```
return;
```

```
}
int res = n;
for (int s = 0; sum[s] + S <= sum[n]; s++) {
```

9

```
// 利用二分搜索求出t
int t = lower_bound(sum + s, sum + n, sum[s] + S) - sum; res = min(res, t - s);
}
printf("%d\n", res);
```

10

```
}
```

这个算法的复杂度是 $O(n \log n)$，虽然足以解决这个问题，但我们还可以更加高效地求解。我们设

以 as 开始总和最初大于 S 时的连续子序列为 $as + \ldots + at$

11

□1，这时

5 1 3 5 10 7 4 9 2 8

5 1 3 5 10 7 4 9 2 8

5 1 3 5 10 7 4 9 2 8

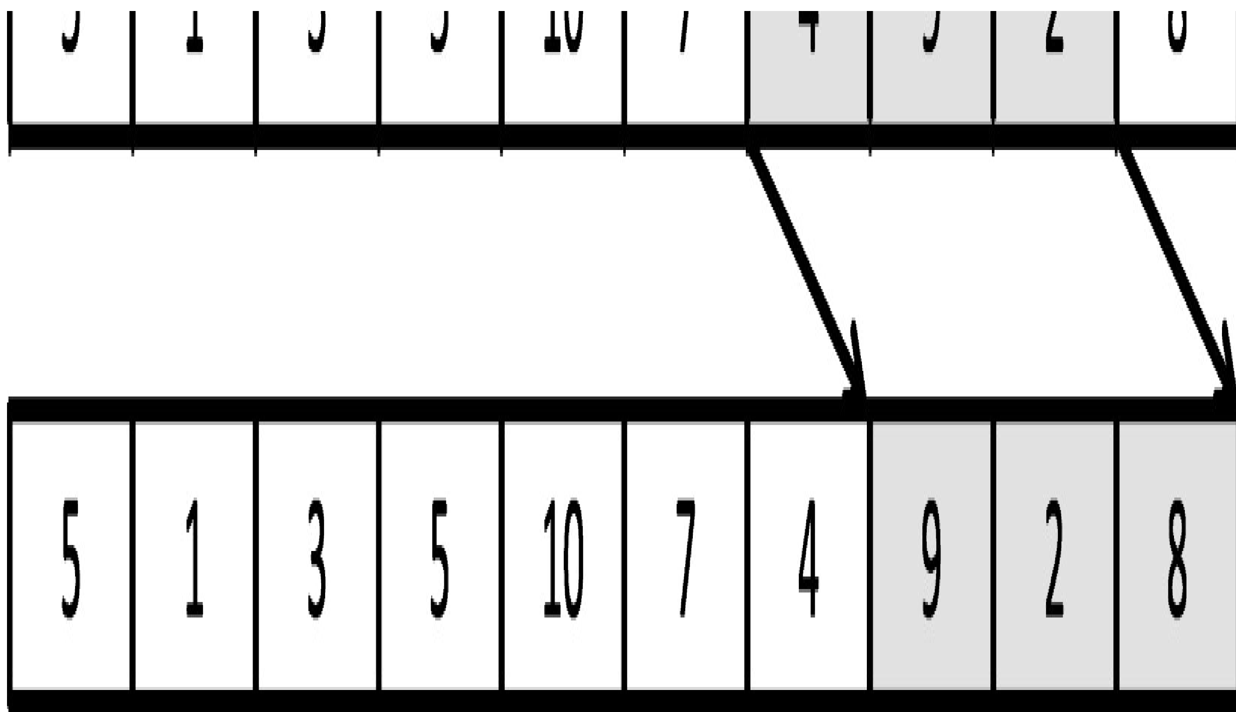
5 1 3 5 10 7 4 9 2 8

5 1 3 5 10 7 4 9 2 8

5 1 3 5 10 7 4 9 2 8

5 1 3 5 10 7 4 9 2 8

5 1 3 5 10 7 4 9 2 8



$a_{s+1}+\ldots+a_{t-2}<a_s+\ldots+a_{t-2}<S$

所以从 a_{s+1}开始总和最初超过 S 的连续子序列如果是 $a_{s+1}+\ldots+a_{t'-1}$ 的话，则必然有 $t\le t'$ 。利用这一性

质便可以设计出如下算法：

(1) 以 $s=t=\text{sum}=0$ 初始化。

(2) 只要依然有 $\text{sum}<S$ ，就不断将 sum 增加 a_t ，并将 t 增加 1。

(3) 如果(2)中无法满足 $\text{sum}\ge S$ 则终止。否则的话，更新 $\text{res}=\min(\text{res}, t-s)$。

(4) 将 sum 减去 a_s ， s 增加 1 然后回到(2)。

对于这个算法，因为 t 最多变化 n 次，因此只需 $O(n)$的复杂度就可以求解这个问题了。

```
void solve() {
```

```
int res = n + 1;
int s = 0, t = 0, sum = 0;
for (;;) {
while (t < n && sum < S) {
sum += a[t++];
}
if (sum < S) break;
res = min(res, t - s);
sum -= a[s++];
}
if (res > n) {
// 解不存在
res = 0;
}
printf("%d\n", res);
}
```

样例数据1对应的区间的变化



像这样反复地推进区间的开头和末尾，来求取满足条件的最小区间的方法被称为尺取法。

1

Jessica’s Reading Problem （POJ No.3320）

为了准备考试，Jessica 开始读一本很厚的课本。要想通过考试，必须把课本中所有的知识点

2

都掌握。这本书总共有 P 页，第 i 页恰好有一个知识点 ai （每个知识点都有一个整数编号）。

全书中同一个知识点可能会被多次提到，所以她希望通过阅读其中连续的一些页把所有的知

识点都覆盖到。给定每页写到的知识点，请求出要阅读的最少页数。

3

限制条件

□ $1 \le P \le 106$

4

样例

输入

P = 5

5

a = {1, 8, 8, 8, 1}

输出

2 (只要阅读第1页和第2页即可)

6

我们假设从某一页 s 开始阅读，为了覆盖所有的知识点需要阅读到 t 。这样的话可以知道如果从 $s+1$

开始阅读的话，那么必须阅读到 $t' \geq t$ 页为止。由此这题也可以使用尺取法。

7

在某个区间[s, t]已经覆盖了所有的知识点的情况下，下一个区间[$s+1, t'$]($t' \geq t$)要如何求出呢?

所有的知识点都被覆盖□每个知识点出现的次数都不小于1

8

由以上的等价关系，我们可以用二叉树等数据结构来存储[s, t]区间上每个知识点的出现次数，这

样把最开头的页 s 去除后便可以判断[$s+1, t$]是否满足条件。

从区间的最开头把 s 取出之后，页 s 上书写的知识点的出现次数就要减一，如果此时这个知识点的

9

出现次数为0了，在同一个知识点再次出现前，不停将区间末尾 t 向后推进即可。每次在区间末尾

追加页 t 时将页 t 上的知识点的出现次数加1，这样就完成了下一个区间上各个知识点出现次数的更

新，通过重复这一操作可以以 $O(P \log P)$的复杂度求出最小的区间。

10

```
// 输入

int P;

int a[MAX_P];

void solve() {
```

11

```
// 计算全部知识点的总数
```




```
set<int> all;

for (int i = 0; i < P; i++) {

all.insert(a[i]);

}
```

```
int n = all.size();

// 利用尺取法来求解

int s = 0, t = 0, num = 0;

map<int, int> count; // 知识点→出现次数的映射

int res = P;

for (;;) {

while (t < P && num < n) {

if (count[a[t++]]++ == 0) {

// 出现新的知识点

num++;

}

}

if (num < n) break;

res = min(res, t - s);

if (--count[a[s++]] == 0) {

// 某个知识点的出现次数为0了

num--;

}

}

printf("%d\n", res);
```

}

3.2.2 反转（开关问题）

Face The Right Way （POJ No. 3276）

N 头牛排成了一列。每头牛或者向前或者向后。为了让所有的牛都面向前方，农夫约翰买了

一台自动转向的机器。这个机器在购买时就必须设定一个数值 K ，机器每操作一次恰好使 K

头连续的牛转向。请求出为了让所有的牛都能面向前方需要的最少的操作次数 M 和对应的

最小的 K 。

限制条件

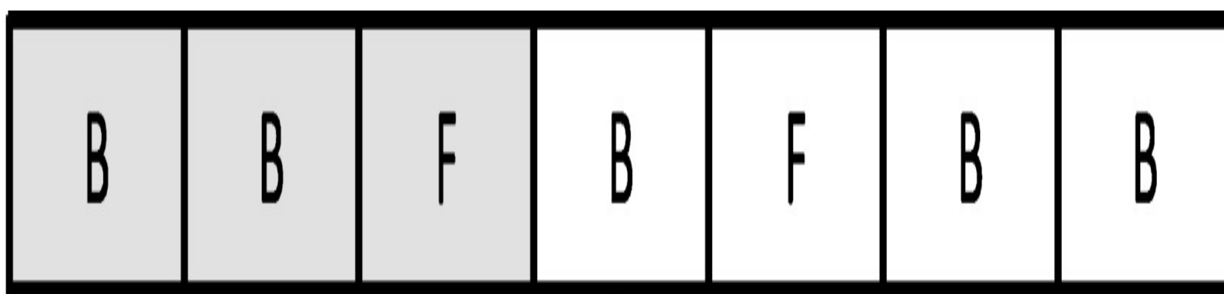
 $1 \leq N \leq 5000$

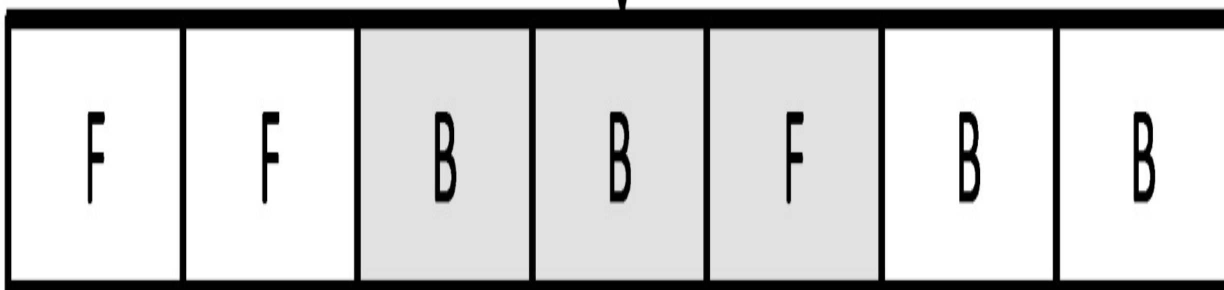
样例

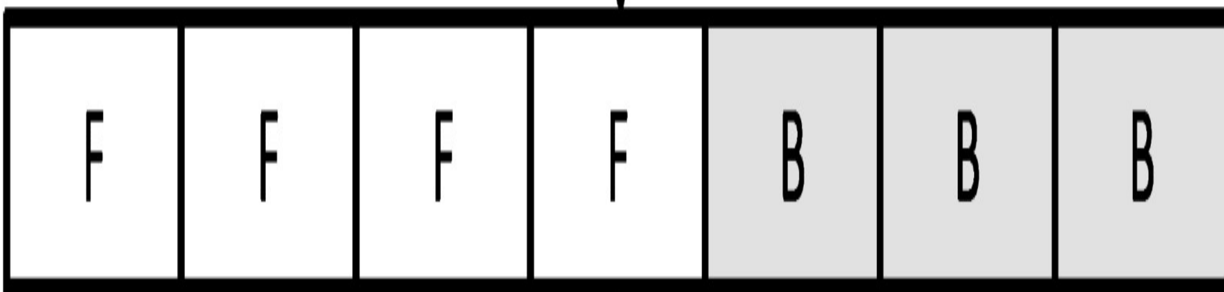
输入

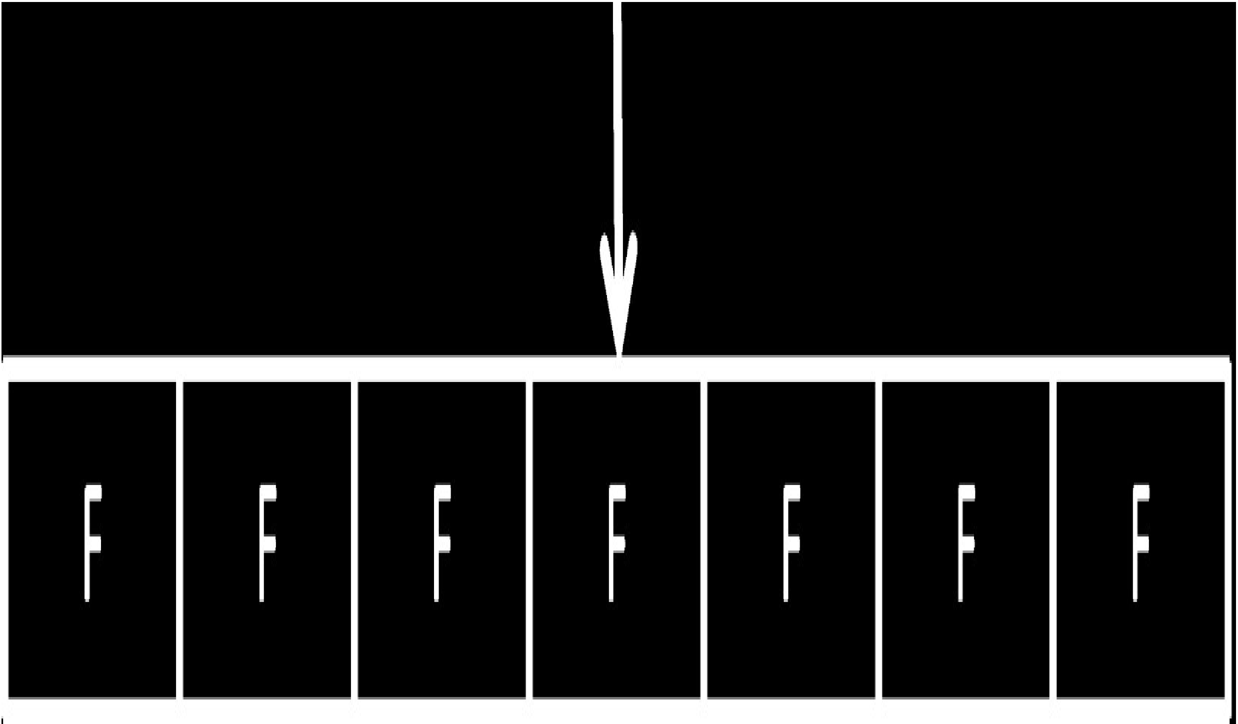
N = 7

BBFBFBB (F：面向前方，B：面向后方)


B
B
F
B
F
B
B


F
F
B
B
F
B
B


F
F
F
F
B
B
B

输出

1

K = 3

M = 3

（先反转1~3号的三头牛，然后再反转3~5号，最后反转5~7号）

2

3

4

K=3时的解法

5

首先我们来看看对于一个特定的 K 如何求出让所有的牛面朝前方的最小操作次数。如果把牛的方

向作为状态进行搜索的话，由于状态数有2 N 个，是无法在时限内找出答案的。那么不搜索的话要

6

怎么办呢?

首先，交换区间反转的顺序对结果是没有影响的。此外，可以知道对同一个区间进行两次以上

的反转是多余的。由此，问题就转化成了求需要被反转的区间的集合。于是我们先考虑一下最

7

左端的牛。包含这头牛的区间只有一个，因此如果这头牛面朝前方，我们就能知道这个区间不

需要反转。

反之，如果这头牛面朝后方，对应的区间就必须进行反转了。而且在此之后这个最左的区间就再

8

也不需要考虑了。这样一来，通过首先考虑最左端的牛，问题的规模就缩小了1。不断地重复下

去，就可以无需搜索求出最少所需的反转次数了。

此外，通过上面的分析可以知道，忽略掉对同一个区间重复反转这类多余操作之后，只要存在让

9

所有的牛都朝前的方法，那么操作就和顺序无关可以唯一确定了。

这个算法的复杂度又如何呢？首先我们需要对所有的 K 都求解一次，对于每个 K 我们都要从最左

端开始来考虑 N 头牛的情况。此时最坏情况下需要进行 N □ K +1次的反转操作，而每次操作又要反

10

转 K 头牛，于是总的复杂度就是 $O(N3)$。这样的话还不足以在时限内解决问题。但是区间反转的部

分还是很容易进行优化的。

$f[i]$:= 区间[i , i + K □1]进行了反转的话则为1，否则为0

11

这样，在考虑第 i 头牛时，如果

i 1

□

□

$f[i]$ 为奇数的话，则这头牛的方向与起始方向是相反的，

j □ i □ K 1

□

否则方向不变。由于

i

i 1

□

□ $f[j]$ □ □ $f[j]$□ $f[i]$□ $f[i$ □ K □$1]$

i □ i

(

K

+1)-

1

+

i □ i □ K 1

□

所以这个和每一次都可以用常数时间计算出来，复杂度就降为了 $O(N$
2)，能在时限内解决了。

```
// 输入

int N;

int dir[MAX_N]; // 牛的方向(0:F, 1:B)

int f[MAX_N]; // 区间[i,i+K-1]是否进行反转

// 固定K，求对应的最小操作回数

// 无解的话则返回-1

int calc(int K) {

memset(f, 0, sizeof(f));

int res = 0;

int sum = 0; // f的和
```

```
for (int i = 0; i + K <= N; i++) {

// 计算区间[i,i+K-1]

if ((dir[i] + sum) % 2 != 0) {

// 前端的牛面朝后方

res++;

f[i] = 1;

}

sum += f[i];

if (i - K + 1 >= 0) {

sum -= f[i - K + 1];

}

}

// 检查剩下的牛是否有面朝后方的情况

for (int i = N - K + 1; i < N; i++) {

if ((dir[i] + sum) % 2 != 0) {

// 无解

return -1;

}

if (i - K + 1 >= 0) {

sum -= f[i - K + 1];
```

```
}
}
return res;
}
void solve() {
int K = 1, M = N;
for (int k = 1; k <= N; k++) {
int m = calc(k);
```




```
if (m >= 0 && M > m) {
```

1

```
M = m;
K = k;
}
}
printf("%d %d\n", K, M);
```

}

2

Fliptile （POJ No.3279）①

3

农夫约翰知道聪明的牛产奶多。于是为了提高牛的智商他准备了如下游戏。有一个 $M \times N$ 的

格子，每个格子可以翻转正反面，它们一面是黑色，另一面是白色。黑色的格子翻转后就是

白色，白色的格子翻转过来则是黑色。游戏要做的就是把所有的格子都翻转成白色。不过因

4

为牛蹄很大，所以每次翻转一个格子时，与它上下左右相邻接的格子也会被翻转。因为翻格

子太麻烦了，所以牛都想通过尽可能少的次数把所有格子都翻成白色。现在给定了每个格子

的颜色，请求出用最小步数完成时每个格子翻转的次数。最小步数的解有多个时，输出字典

序最小的一组。解不存在的话，则输出 IMPOSSIBLE。

5

限制条件

 $1 \le M, N \le 15$

6

样例

输入

7

M = 4

N = 4

每个格子的颜色如下(0表示白色，1表示黑色)

1 0 0 1

8

0 1 1 0

0 1 1 0

1 0 0 1

9

输出

0 0 0 0

1 0 0 1

1 0 0 1

0 0 0 0

10

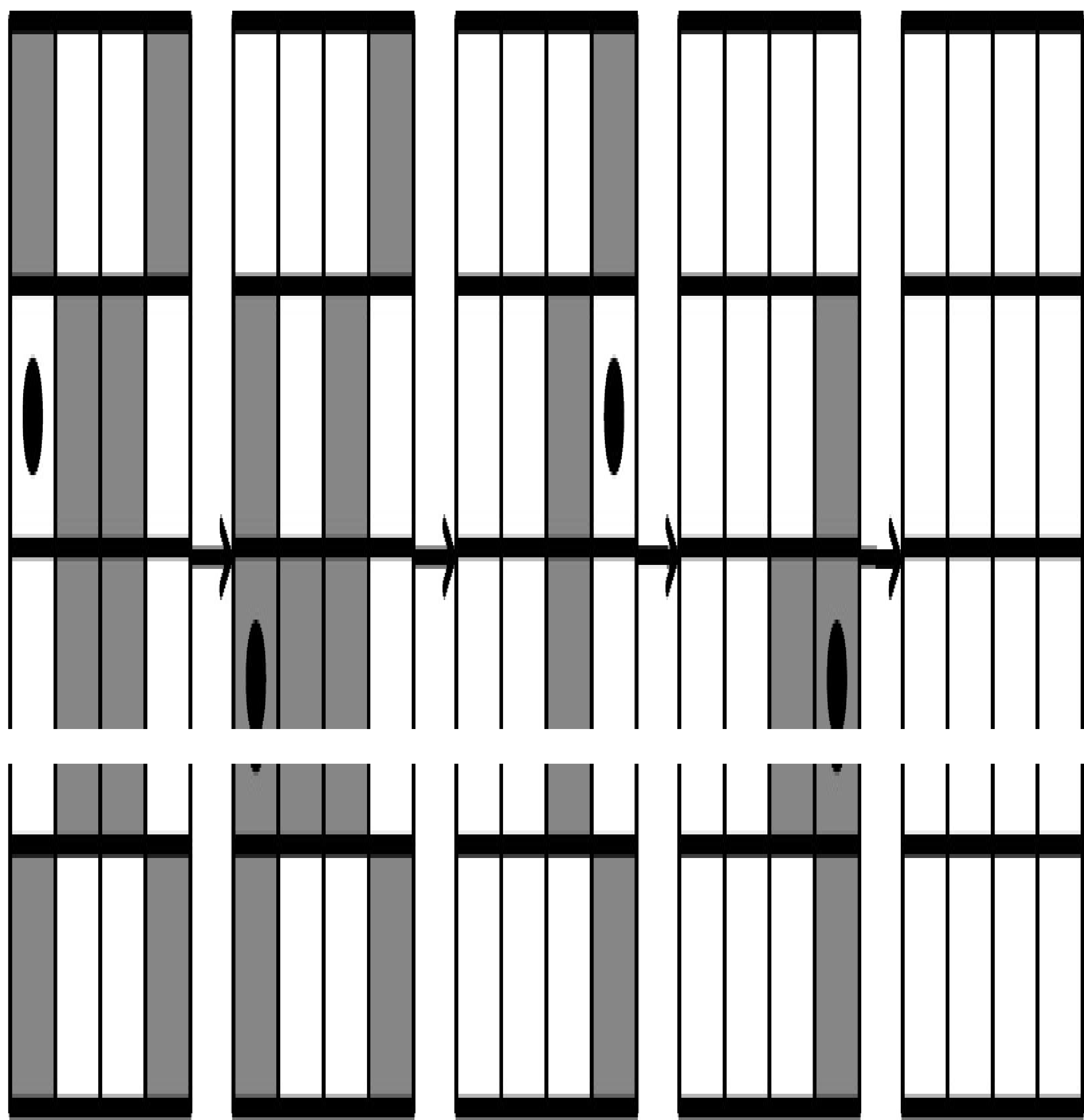
---

① 这道题的模型常被称为开关问题或关灯问题，较早见于Extended Lights Out（Greater New York 2002）这道题。高斯消

元法也可以用于求解一组可行解，并且通过这些分析可以知道，自由变元的个数不会超过 N 个，所以也可以用于求解

11

最优解。——译者注



解法示意

首先，同一个格子翻转两次的话就会恢复原状，所以多次翻转是多余的。此外，翻转的格子的集

合相同的话，其次序是无关紧要的。因此，总共有2^{NM}种翻转的方法。不过这个解空间太大了，

我们需要想出更有效的办法。

让我们再回顾一下前面的问题。在那道题中，让最左端的牛反转的方法只有1种，于是用直接判

断的方法确定就可以了。同样的方法在这里还行得通吗?

不妨先看看最左上角的格子。在这里，除了翻转(1,1)之外，翻转(1,2)和(2,1)也可以把这个格子翻

转，所以像之前那样直接确定的办法行不通。

于是不妨先指定好最上面一行的翻转方法。此时能够翻转(1,1)的只剩下(2,1)了，所以可以直接判

断(2,1)是否需要翻转。类似地(2,1)~(2, N)都能这样判断，如此反复下去就可以确定所有格子的翻

转方法。最后(M,1)~(M, N)如果并非全为白色，就意味着不存在可行的操作方法。

像这样，先确定第一行的翻转方式，然后可以很容易判断这样是否存在解以及解的最小步数是多

少，这样将第一行的所有翻转方式都尝试一次就能求出整个问题的最小步数。这个算法中最上面

一行的翻转方式共有2^N种，复杂度为$O(MN2^N)$。

```
// 邻接的格子的坐标

const int dx[5] = {-1, 0, 0, 0, 1};
```

```
const int dy[5] = {0, -1, 0, 1, 0};
// 输入
int M, N;
int tile[MAX_M][MAX_N];
int opt[MAX_M][MAX_N]; // 保存最优解
int flip[MAX_M][MAX_N]; // 保存中间结果
// 查询(x,y)的颜色
int get(int x, int y) {
int c = tile[x][y];
for (int d = 0; d < 5; d++) {
int x2 = x + dx[d], y2 = y + dy[d];
if (0 <= x2 && x2 < M && 0 <= y2 && y2 < N) {
c += flip[x2][y2];
}
}
return c % 2;
```



```
}
```

1

```
// 求出第1行确定情况下的最小操作次数
```

}

```
// 不存在解的话返回-1
int calc() {
// 求出从第2行开始的翻转方法
for (int i = 1; i < M; i++) {
2
for (int j = 0; j < N; j++) {
if (get(i - 1, j) != 0) {
// (i-1,j)是黑色的话，则必须翻转这个格子
flip[i][j] = 1;
}
3
}
}
// 判断最后一行是否全白
for (int j = 0; j < N; j++) {
4
if (get(M - 1, j) != 0) {
// 无解
return -1;
}
```

```
}
5
// 统计翻转的次数
int res = 0;
for (int i = 0; i < M; i++) {
for (int j = 0; j < N; j++) {
6
res += flip[i][j];
}
}
return res;
}
7
void solve() {
int res = -1;
// 按照字典序尝试第一行的所有可能性
for (int i = 0; i < 1 << N; i++) {
8
memset(flip, 0, sizeof(flip));
for (int j = 0; j < N; j++) {
```

```
flip[0][N - j - 1] = i >> j & 1;
}
int num = calc();
```

9

```
if (num >= 0 && (res < 0 || res > num)) {
res = num;
memcpy(opt, flip, sizeof(flip));
}
}
```

10

```
if (res < 0) {
// 无解
printf("IMPOSSIBLE\n");
} else {
```

11

```
for (int i = 0; i < M; i++) {
```



```
for (int j = 0; j < N; j++) {
printf("%d%c", opt[i][j], j + 1 == N ? '\n' : ' ');
}
```

}

}

}

专栏 集合的整数表示

前面的代码里，为了尝试第一行的所有可能性，使用了集合的整数表现。在程序中表示集合

的方法有很多种，当元素数比较少时，像这样用二进制码来表示比较方便。集合{0,1,…, n 1}

的子集 S 可以用如下方式编码成整数。

$f (S)$ 2 i

i S

像这样表示之后，一些集合运算可以对应地写成如下方式。

 空集：..0

 只含有第 i 个元素的集合{ i }：.................................1<<i

 含有全部 n 个元素的集合{0,1,…, n 1}： ....(1<<n)-1

 判断第 i 个元素是否属于集合 S：............................if (S>>i&1)

 向集合中加入第 i 个元素 S {

∪ i }：..........................S|1<<i

 从集合中去除第 i 个元素 S \{ i }：.............................S&~(1<<i)

 集合 S 和 T 的并集 S ∪ T： ....S|T

□ 集合 *S* 和 *T* 的交集 *S* ∩ *T*： ....S&T

此外，想要将集合{0,1,…, *n* □1}的所有子集枚举出来的话，可以像下面这样书写

```
for (int S = 0; S < 1 << n; S++) {

// 对子集的处理

}
```

按照这个顺序进行循环的话， *S* 便会从空集开始，然后按照{0}、{1}、{0,1}、…、{0,1,…, *n* □1}

的升序顺序枚举出来。

接下来介绍一下如何枚举某个集合 sup 的子集。这里 sup 是一个二进制码，其本身也是某

个集合的子集。例如给定了 01101101 这样的集合，要将 01100000 或者 00101101 等子集枚举

出来。前面是从 0 开始不断加 1 来枚举出了全部的子集。此时，sub+1 并不一定是 sup 的

子集。而(sub+1)&sup 虽然是 sup 的子集，可是很有可能依旧是 sub，没有任何改变。

所以我们要反过来，从 sup 开始每次减 1 直到 0 为止。由于 sub-1 并不一定是 sup 的子集，

所以我们把它与 sup 进行按位与&。这样的话就可以将 sup 所有的子集按照降序列举出来。

(sub-1)&sup 会忽略 sup 中的 0 而从 sub 中减去 1。


```
int sub = sup;
```

1

```
do {

// 对子集的处理

sub = (sub - 1) & sup;

} while(sub != sup); // 处理完0之后，会有-1&sup=sup
```

2

最后我们介绍一下枚举$\{0,1,\ldots,n-1\}$所包含的所有大小为 k 的子集的方法。通过使用位运算我

们可以像如下代码所示简单地按照字典序升序地枚举出所有满足条件的二进制码。

```
int comb = (1 << k) - 1;

while (comb < 1 << n) {
```

3

```
// 这里进行针对组合的处理

int x = comb & -comb, y = comb + x;

comb = ((comb & ~y) / x >> 1) | y;

}
```

4

按照字典序的话，最小的子集是(1<<k)-1，所以用它作为初始值。现在我们求出 comb 其

后的二进制码。例如 0101110 之后的是 0110011，0111110 之后的是 1001111。下面是求出 comb 下一个二进制码的方法。

5

(1) 求出最低位的 1 开始的连续的 1 的区间（0101110→0001110）

(2) 将这一区间全部变为 0，并将区间左侧的那个 0 变为 1（0101110→0110000）

(3) 将第(1)步里取出的区间右移，直到剩下的 1 的个数减少了 1 个（0001110→0000011）

(4) 将第(2)步和第(3)步的结果按位取或（0110000|0000011=0110011）

6

对于非零的整数，x&(-x)的值就是将其最低位的 1 独立出来后的值。这是由于计算机中负

数采用二进制补码表示，-x 实际上对应于(~x)+1（将 x 按位取反然后加上 1）。

7

x

x 的二进制

$-x$ 的二进制

$x \& -x$

1 0001

1111

0001

2 0010

1110

0010

3 0011

1101

0001

4 0100

1100

0100

5 0101

1011

0001

8

6 0110

1010

0010

7 0111

1001

0001

将最低位的 1 取出后，设它为 x。那么通过计算 y=comb+x，就将 comb 从最低位的 1 开始

9

的连续的 1 都置 0 了。我们来比较一下~y 和 comb。在 comb 中加上 x 后没有变化的位，在

~y 中全都取相反的值。而最低位 1 开始的连续区间在~y 中依然是 1，区间左侧的那个 0 在

~y 中也依然是 0。于是通过计算 z=comb&~y 就得到了最低位 1 开始的连续区间。比如如果

10

comb=0101110，则 x =0000010， y =0110000， z =0001110。

同时，y 也恰好是第(2)步要求的值。那么首先将 z 不断右移，直到最低位为 1，这通过计算

z/x 即可完成。这样再将 z/x 右移 1 位就得到了第(3)步要求的值。这样我们就求得了 comb

11



之后的下一个二进制列。因为是从 n 个元素的集合中进行选择，所以 comb 的值不能大于等

于 1<<n。如此一来，就完成了大小为 k 的所有子集的枚举。

除上述例子之外，还可以利用位运算完成满足其他条件的集合的枚举，例如不包含相邻元素

的集合等。

3.2.3 弹性碰撞

Physics Experiment （POJ No.3684）

用 N 个半径为 R 厘米的球进行如下实验。

在 H 米高的位置设置了一个圆筒，将球垂直放入（从下向上数第 i 个球的底端距离地面高度

为 $H+2R$ ）。实验开始时最下面的球开始掉落，此后每一秒又有一个球开始掉落。不计空气阻

力，并假设球与球或地面间的碰撞是弹性碰撞。

请求出实验开始后 T 秒种时每个球底端的高度。假设重力加速度为 g =10m/s2。

限制条件

 $1 \le N \le 100$

 $1 \le H \le 10000$

 $1 \le R \le 100$

 $1 \le T \le 10000$

样例 1

输入

N = 1

H = 10

R = 10

T = 100

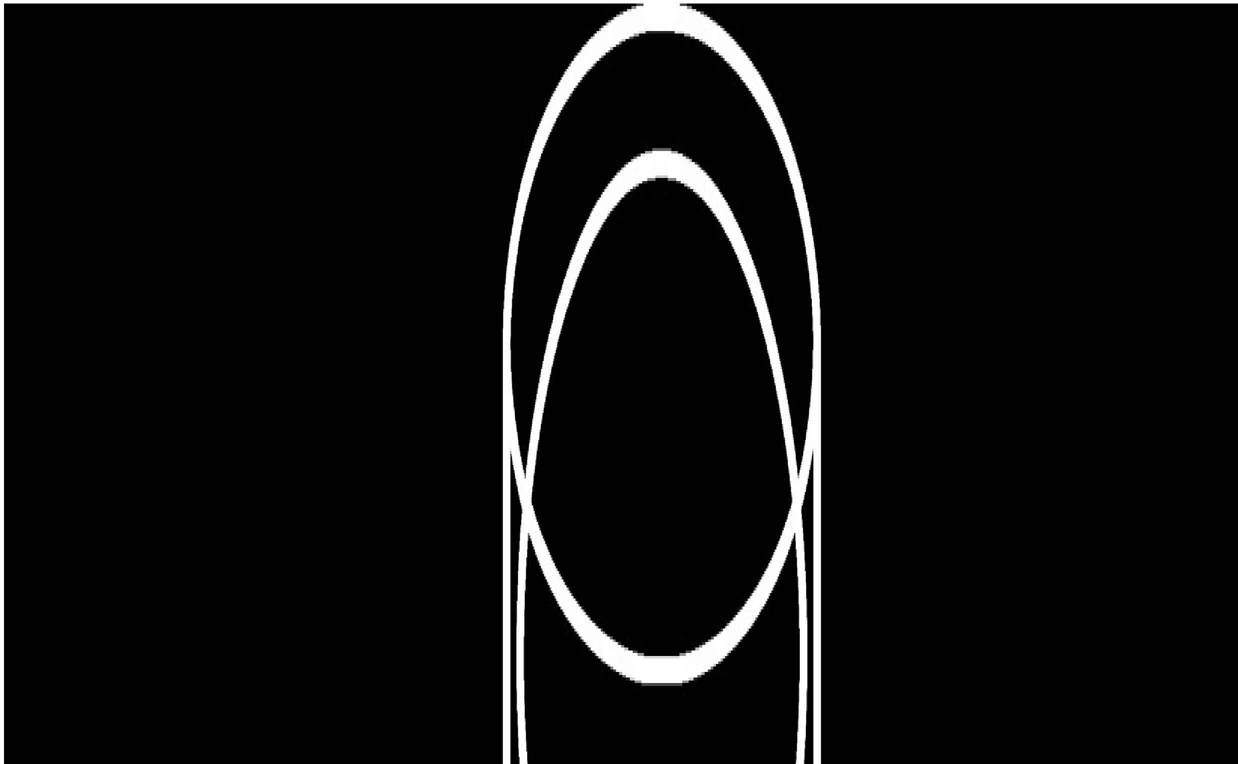
输出

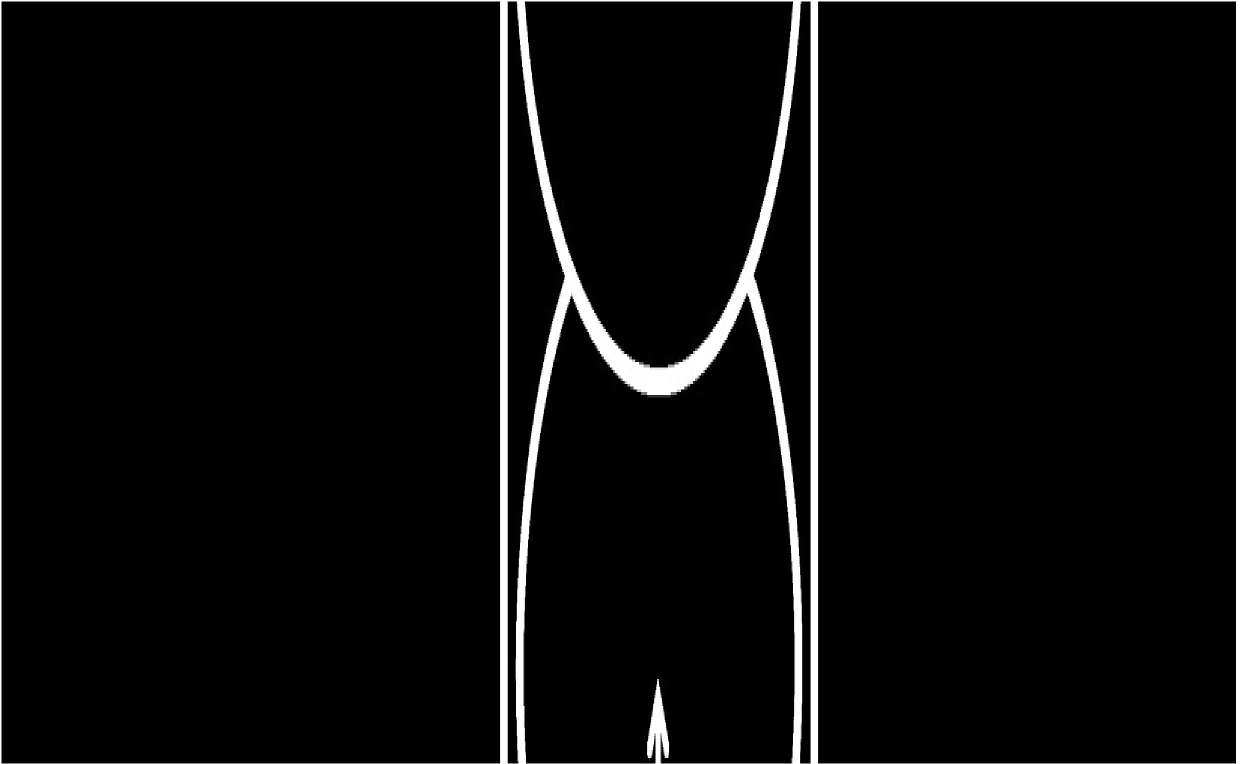
4.95

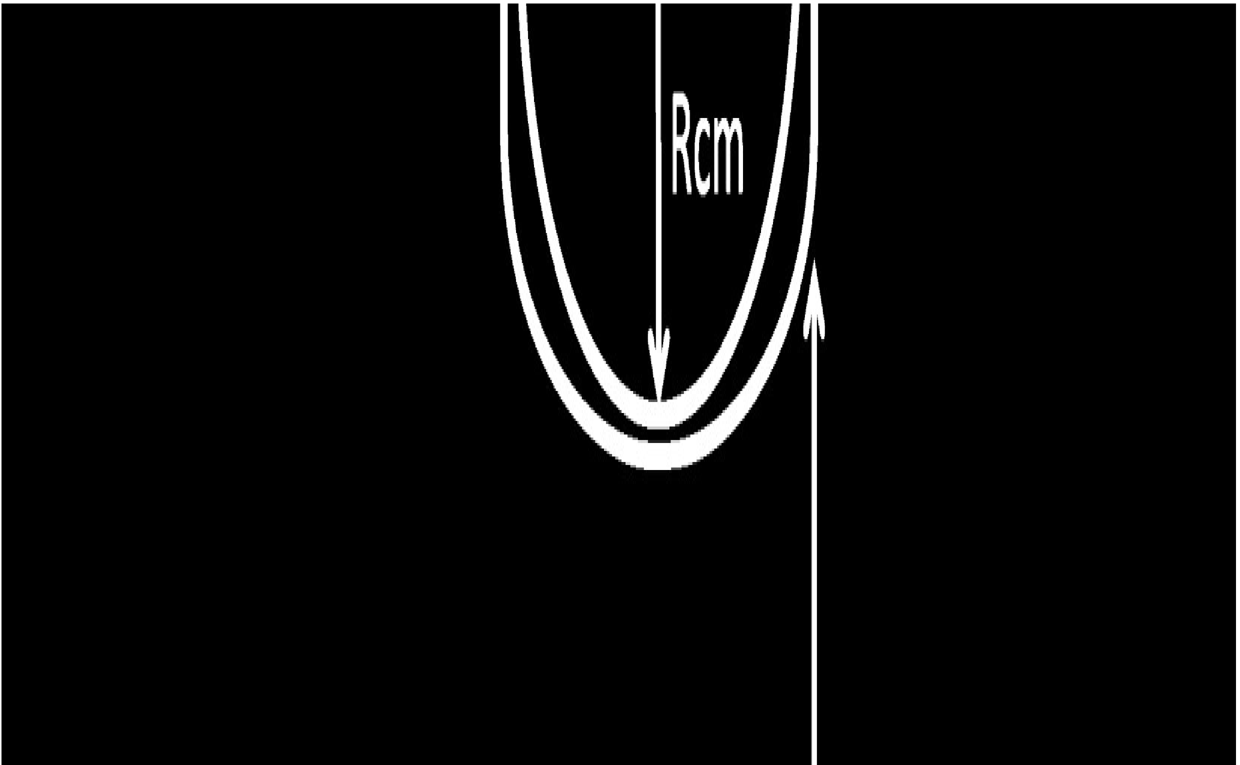
样例 2

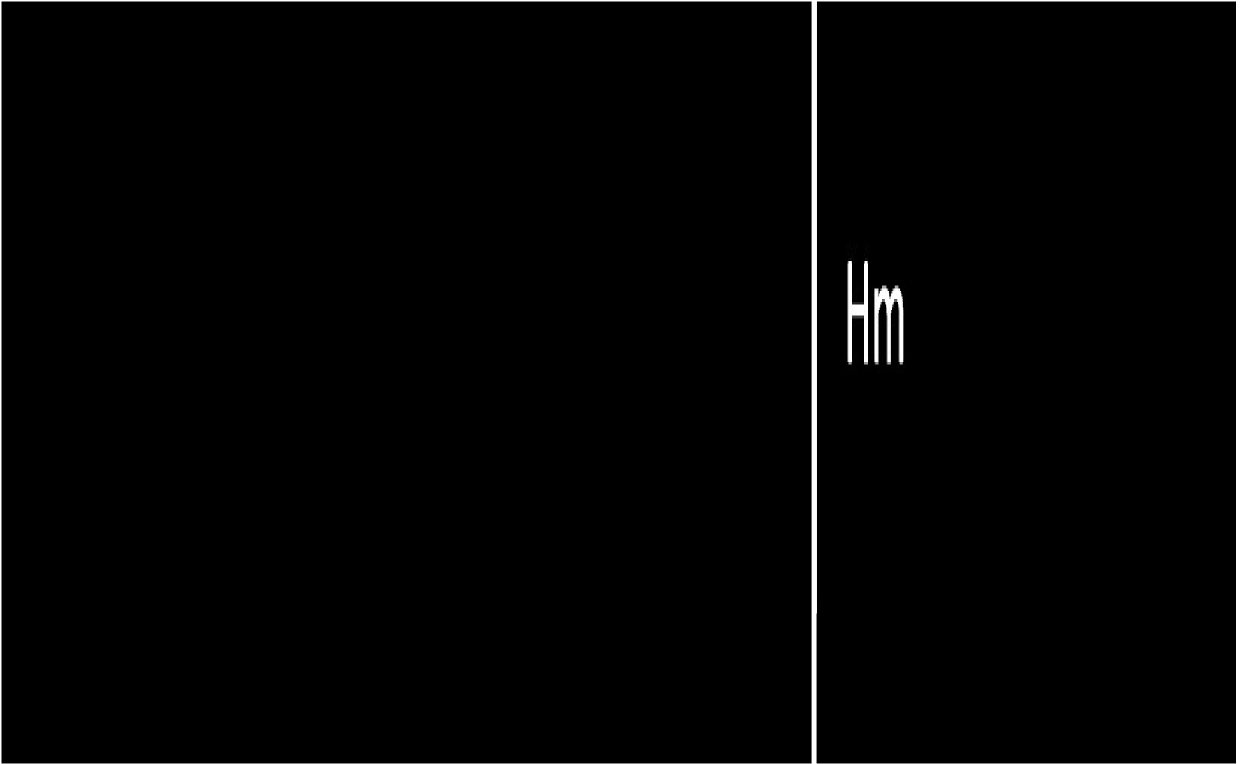
输入

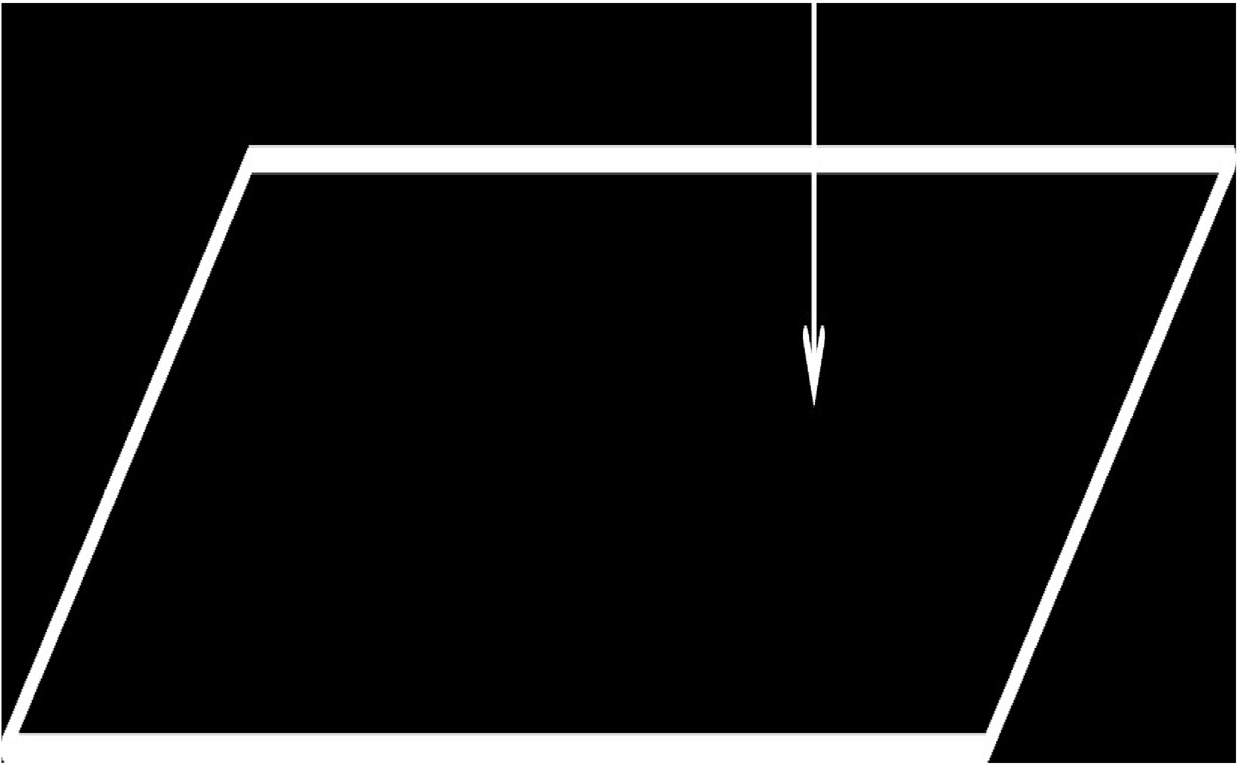
N = 2






Rcm


Hm



$$y=\begin{cases} H-\dfrac{1}{2}g(T-kt)^2 & (\text{是 } k \text{ 偶数时}) \\ H-\dfrac{1}{2}g(kt+t-T)^2 & (\text{是 } k \text{ 奇数时}) \end{cases}$$

H = 10

1

R = 10

T = 100

输出

2

4.95 10.20

3

4

5

初始状态

首先考虑一下只有一个球的情形。这时只是单纯的物理问题。从高为 H 的位置下落的话需要花费

6

的时间是

2 H

t

7

g

这样的话，在时刻 T 时，令 k 为满足 $kt \leq T$ 的最大整数，那么

8

9

接下来再考虑多个球的情形。乍一看，因为多个球之间会有碰撞，必须对物理运动进行模拟，事

实上并没有这个必要。我们来回忆一下此前热身题目“Ants”。在那道题目中两只蚂蚁相遇后并

不是各自折返，而是当作擦身而过继续走下去，于是就将问题简化了。

10

这里的问题可以用同样方法思考。首先先来考虑一下 $R=0$的情况。如果认为所有的球都是一样的，

就可以无视它们的碰撞，视为直接互相穿过继续运动。由于在有碰撞时球的顺序不会发生改变，

11

所以忽略碰撞，将计算得到的坐标进行排序后，就能知道每个球的最终位置。那么， $R>0$时要怎



么办呢？这种情况下的处理方法基本相同，对于从下方开始的第 i 个球，在按照 $R=0$计算的结果上

加上2 Ri 就好了。

```
const double g = 10.0; // 重力加速度

// 输入

int N, H, R, T;
```

```
double y[MAX_N]; // 球的最终位置

// 求出T时刻球的位置

double calc(int T) {

if (T < 0) return H;

double t = sqrt(2 * H / g);

int k = (int)(T / t);

if (k % 2 == 0) {

double d = T - k * t;

return H - g * d * d / 2;

} else {

double d = k * t + t - T;

return H - g * d * d / 2;

}

}

void solve() {

for (int i = 0; i < N; i++) {

y[i] = calc(T - i);

}

sort(y, y + N);

for (int i = 0; i < N; i++) {
```

```
printf("%.2f%c", y[i] + 2 * R * i / 100.0, i + 1 == N ? '\n' : ' ');
}
}
```

3.2.4 折半枚举（双向搜索）①

4 Values whose Sum is 0 （POJ No.2785）

给定各有 n 个整数的四个数列 A 、 B 、 C 、 D 。要从每个数列中各取出 1 个数，使四个数的和为

0。求出这样的组合的个数。当一个数列中有多个相同的数字时，把它们作为不同的数字看待。

限制条件

- $1 \le n \le 4000$
- |(数字的值)| ≤ 228

---

① 本节所介绍的折半枚举与传统的双向搜索并不相同，但其思想来源于传统的双向搜索，有时候我们也会用双向搜索来

指代它，特此注明。——译者注

样例

1

输入

n = 6

A = {-45, -41, -36, -36, 26, -32}

B = {22, -27, 53, 30, -38, -54}

2

C = {42, 56, -37, -75, -10, -6}

D = {-16, 30, 77, -46, 62, 45}

输出

3

5(-45-27+42+30=0, 26+30-10-46=0, -32+22+56-46=0, -32+30-75+77=0, -32-54+56+30=0) 这个问题是热身题目“抽签”的复习。从4个数列中选择的话总共有 n^4种情况，所以全都判断一

遍不可行。不过将它们对半分成 AB 和 CD 再考虑，就可以快速解决了。从2个数列中选择的话只有

4

n^2种组合，所以可以进行枚举。先从 A 、 B 中取出 a 、 b 后，为了使总和为0则需要从 C 、 D 中取出

$c + d = -a - b$ 。因此先将从 C 、 D 中取数字的 n^2种方法全都枚举出来，将这些和排好序，这样就可以

运用二分搜索了。这个算法的复杂度是 $O(n^2\log n)$。

5

有时候，问题的规模较大，无法枚举所有元素的组合，但能够枚举一半元素的组合。此时，将问

题拆成两半后分别枚举，再合并它们的结果这一方法往往非常有效。

// 输入

6

```
int n;

int A[MAX_N], B[MAX_N], C[MAX_N], D[MAX_N];

int CD[MAX_N * MAX_N]; // C和D中数字的组合方法

void solve() {
```

7

```
// 枚举从C和D中取出数字的所有方法

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

CD[i * n + j] = C[i] + D[j];

}
```

8

```
}

sort(CD, CD + n * n);

long long res = 0;

for (int i = 0; i < n; i++) {
```

9

```
for (int j = 0; j < n; j++) {

int cd = -(A[i] + B[j]);

// 取出C和D中和为cd的部分

res += upper_bound(CD, CD + n * n, cd) - lower_bound(CD, CD + n * n, cd);
```

}

10

}

printf("%lld\n", res);

}

11



超大背包问题

有重量和价值分别为 w_i , v_i 的 n 个物品。从这些物品中挑选总重量不超过 W 的物品，求所有

挑选方案中价值总和的最大值。

限制条件

- $1 \le n \le 40$

- $1 \le w_i , v_i \le 1015$

- $1 \le W \le 1015$

样例

输入

n = 4

w = {2, 1, 3, 2}

v = {3, 2, 4, 2}

W = 5

输出

7(挑选0、1、3号物品)

这个问题是第二章介绍过的背包问题。不过这次价值和重量都可以是非常大的数值，相比之下 n

比较小。使用DP求解背包问题的复杂度是 $O(nW)$，因此不能用来解决这里的问题。此时我们应该

利用 n 比较小的特点来寻找其他办法。

挑选物品的方法总共有 2^n 种，所以不能直接枚举，但是像前面一样拆成两半之后再枚举的话，因

为每部分只有20个所以是可行的。利用拆成两半后的两部分的价值和重量，我们能求出原先的问

题吗？我们把前半部分中的选取方法对应的重量和价值总和记为 w_1、v_1。这样在后半部分寻找总

重 $w_2 \le W - w_1$ 时使 v_2 最大的选取方法就好了。

因此，我们要思考从枚举得到的 (w_2, v_2) 的集合中高效寻找 $\max\{v_2 | w_2 \le W'\}$ 的方法。首先，显

然我们可以排除所有 $w_2[i] \le w_2[j]$ 并且 $v_2[i] \ge v_2[j]$ 的 j。这一点可以按照 w_2、v_2 的字典序排序后

简单做到。此后剩余的元素都满足 $w_2[i] < w_2[j] \Leftrightarrow v_2[i] < v_2[j]$，要计算 $\max\{v_2 | w_2 \le W'\}$ 的话，只

要寻找满足 $w2[i] \le W'$的最大的 i 就可以了。这可以用二分搜索完成，剩余的元素个数为 M 的话，

一次搜索需要 $O(\log M)$的时间。因为 $M \le 2(n/2)$，所以这个算法总的复杂度是 $O(2(n/2)\, n)$，可以在时

限内解决这个问题。


```
typedef long long ll;
```

1

```
// 输入

int n;

ll w[MAX_N], v[MAX_N];

ll W;
```

2

```
pair<ll, ll> ps[1 << (MAX_N / 2)]; // (重量, 价值) void solve() {

// 枚举前半部分

int n2 = n / 2;
```

3

```
for (int i = 0; i < 1 << n2; i++) {

ll sw = 0, sv = 0;

for (int j = 0; j < n2; j++) {

if (i >> j & 1) {
```

```
sw += w[j];
4
sv += v[j];
}
}
ps[i] = make_pair(sw, sv);
}
5
// 去除多余的元素
sort(ps, ps + (1 << n2));
int m = 1;
for (int i = 1; i < 1 << n2; i++) {
6
if (ps[m - 1].second < ps[i].second) {
ps[m++] = ps[i];
}
}
7
// 枚举后半部分并求解
ll res = 0;
```

```
for (int i = 0; i < 1 << (n - n2); i++) {

ll sw = 0, sv = 0;

for (int j = 0; j < n - n2; j++) {
```

8

```
if (i >> j & 1) {

sw += w[n2 + j];

sv += v[n2 + j];

}

}

if (sw <= W) {
```

9

```
ll tv = (lower_bound(ps, ps + m, make_pair(W - sw, INF)) - 1)->second; res
= max(res, sv + tv);

}

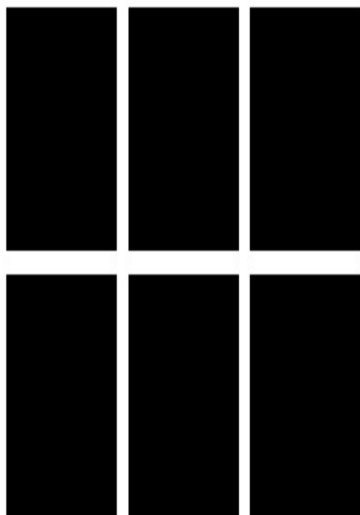
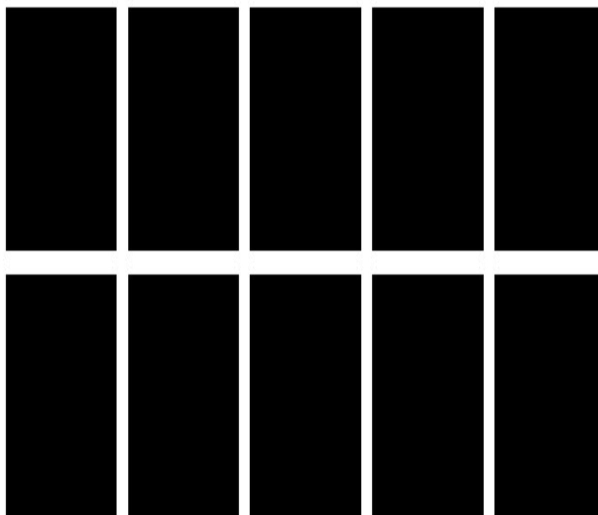
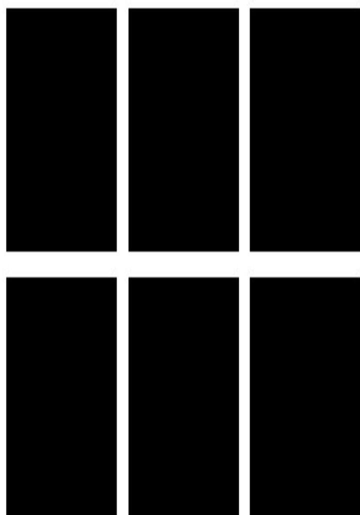
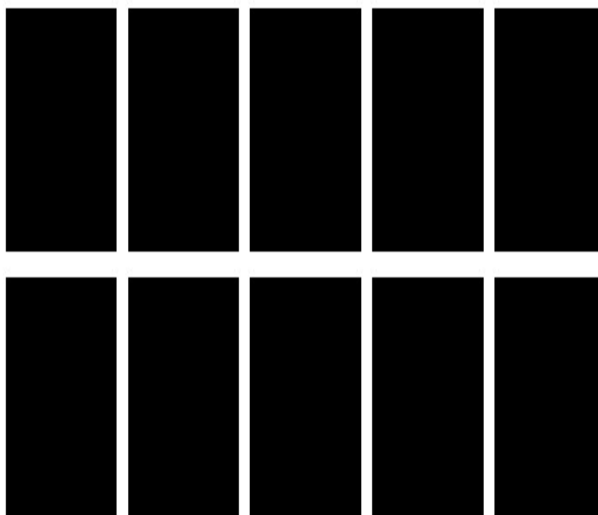
}
```

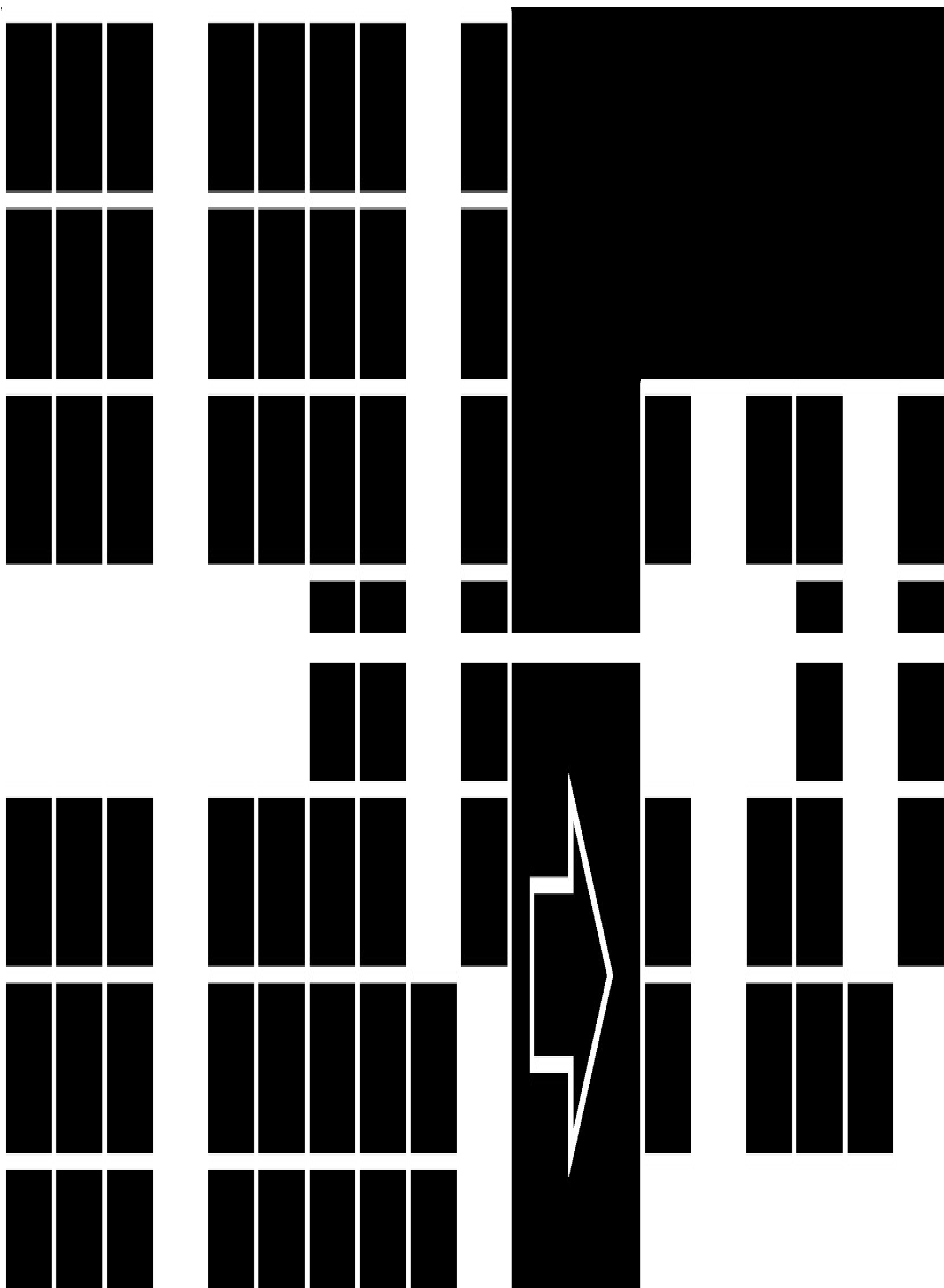
10

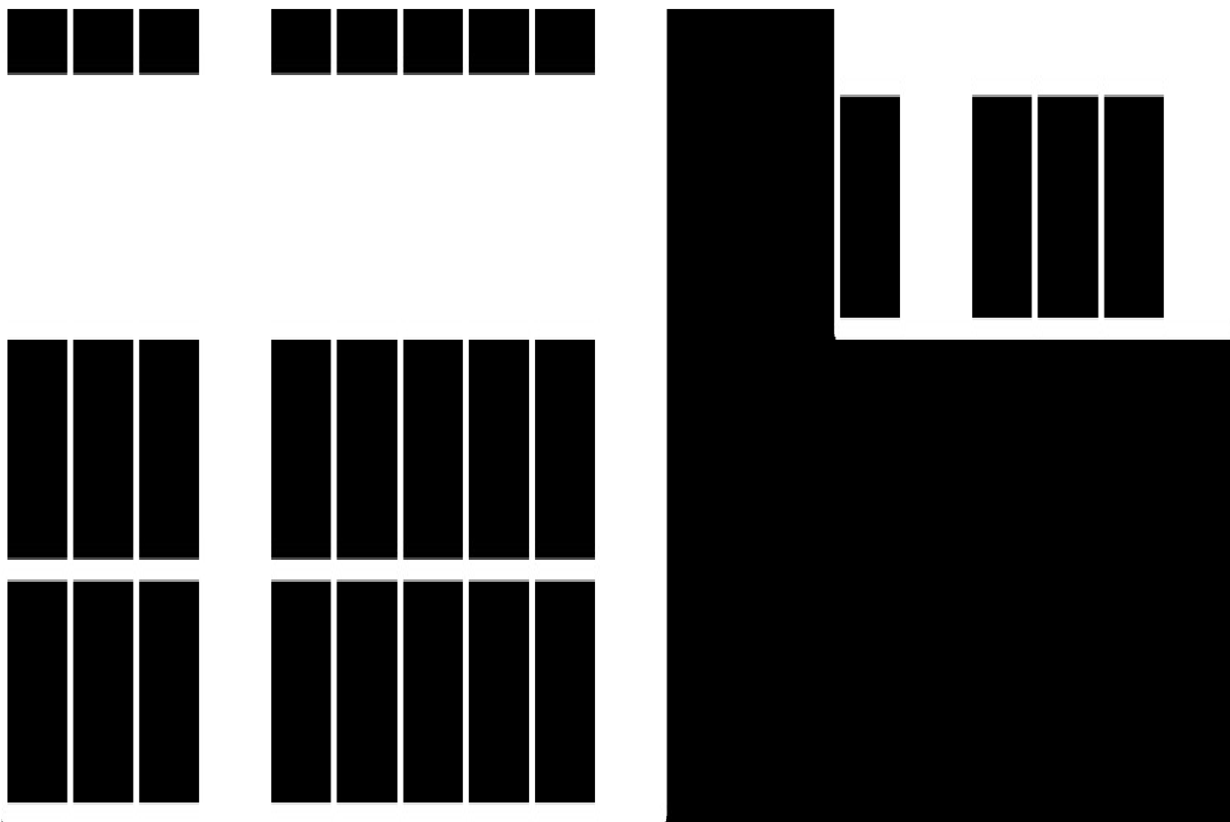
```
printf("%lld\n", res);

}
```

11



3.2.5 坐标离散化

区域的个数

$w \times h$ 的格子上画了 n 条或垂直或水平的宽度为1的直线。求出这些线将格子划分成了多少个区域。

例

限制条件

- $1 \leq w, h \leq 1000000$
- $1 \leq n \leq 500$

样例

w = 10, h = 10, n = 5

x1 = {1, 1, 4, 9, 10}

x2 = {6, 10, 4, 9, 10}

y1 = {4, 8, 1, 1, 6}

y2 = {4, 8, 10, 5, 10}

(对应于前面的例图)

准备好 $w \times h$ 的数组，并记录是否有直线通过，然后参考2.1节利用深度优先搜索求水坑数的方法，

可以求出被分割出的区域个数。但是这个问题中 w 和 h 最大为1000000，所以没办法创建 $w \times h$ 的数

组。因此我们要使用坐标离散化这一技巧。

坐标离散化示例

如上图所示，将前后没有变化的行列消除后并不会影响区域的个数。

1

数组里只需要存储有直线的行列以及其前后的行列就足够了，这样的话大小最多6 n ×6 n 就足够了。

因此就可以创建出数组并利用搜索①求出区域的个数了。

2

```
// 输入

int W, H, N;

int X1[MAX_N], X2[MAX_N], Y1[MAX_N], Y2[MAX_N];

// 填充用
```

```
bool fld[MAX_N * 6][MAX_N * 6];

3

// 对x1和x2进行坐标离散化，并返回离散化之后的宽度

int compress(int *x1, int *x2, int w) {

vector<int> xs;

4

for (int i = 0; i < N; i++) {

for (int d = -1; d <= 1; d++) {

int tx1 = x1[i] + d, tx2 = x2[i] + d;

if (1 <= tx1 && tx1 <= W) xs.push_back(tx1); if (1 <= tx2 && tx2 <= W)
xs.push_back(tx2); 5

}

}

sort(xs.begin(), xs.end());

xs.erase(unique(xs.begin(), xs.end()), xs.end());

6

for (int i = 0; i < N; i++) {

x1[i] = find(xs.begin(), xs.end(), x1[i]) - xs.begin();

x2[i] = find(xs.begin(), xs.end(), x2[i]) - xs.begin();

}

7
```

```
return xs.size();
}
void solve() {
```

8

```
// 坐标离散化
W = compress(X1, X2, W);
H = compress(Y1, Y2, H);
// 填充有直线的部分
memset(fld, 0, sizeof(fld));
```

9

```
for (int i = 0; i < N; i++) {
for (int y = Y1[i]; y <= Y2[i]; y++) {
for (int x = X1[i]; x <= X2[i]; x++) {
fld[y][x] = true;
}
```

10

```
}
}
```

11

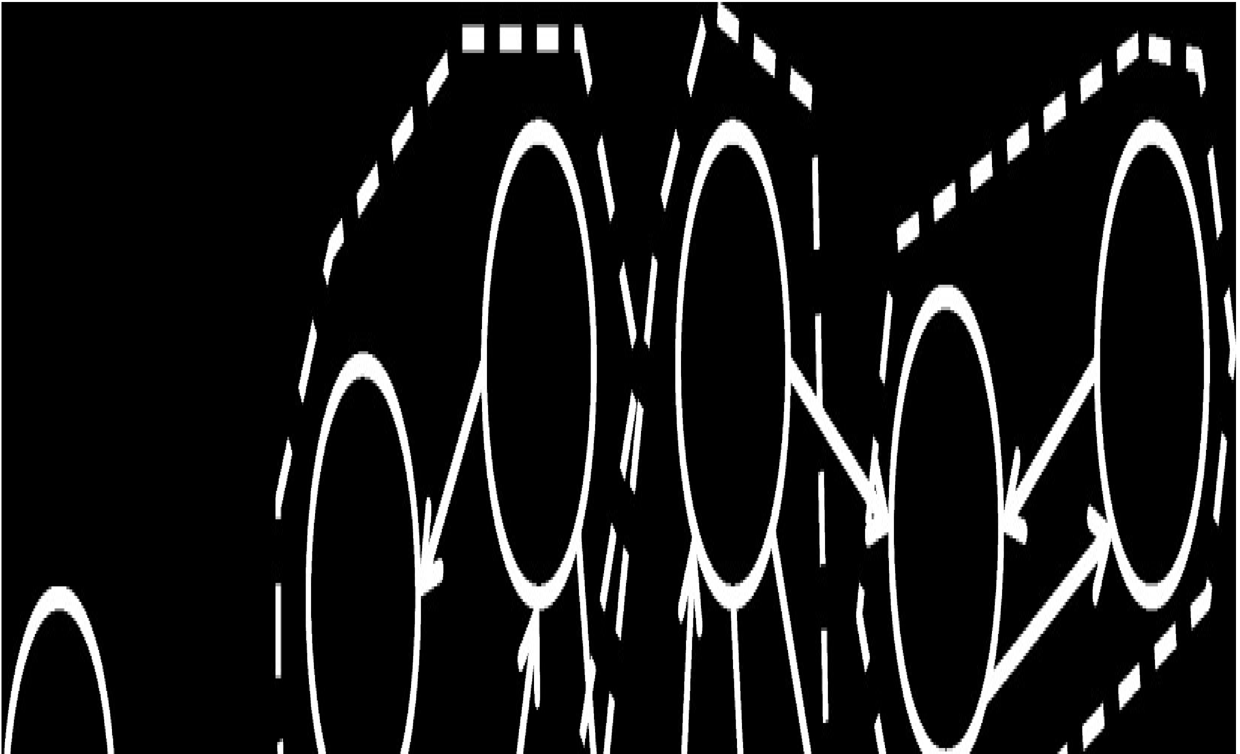
① 区域可能很大，所以用递归函数实现的话可能会栈溢出。

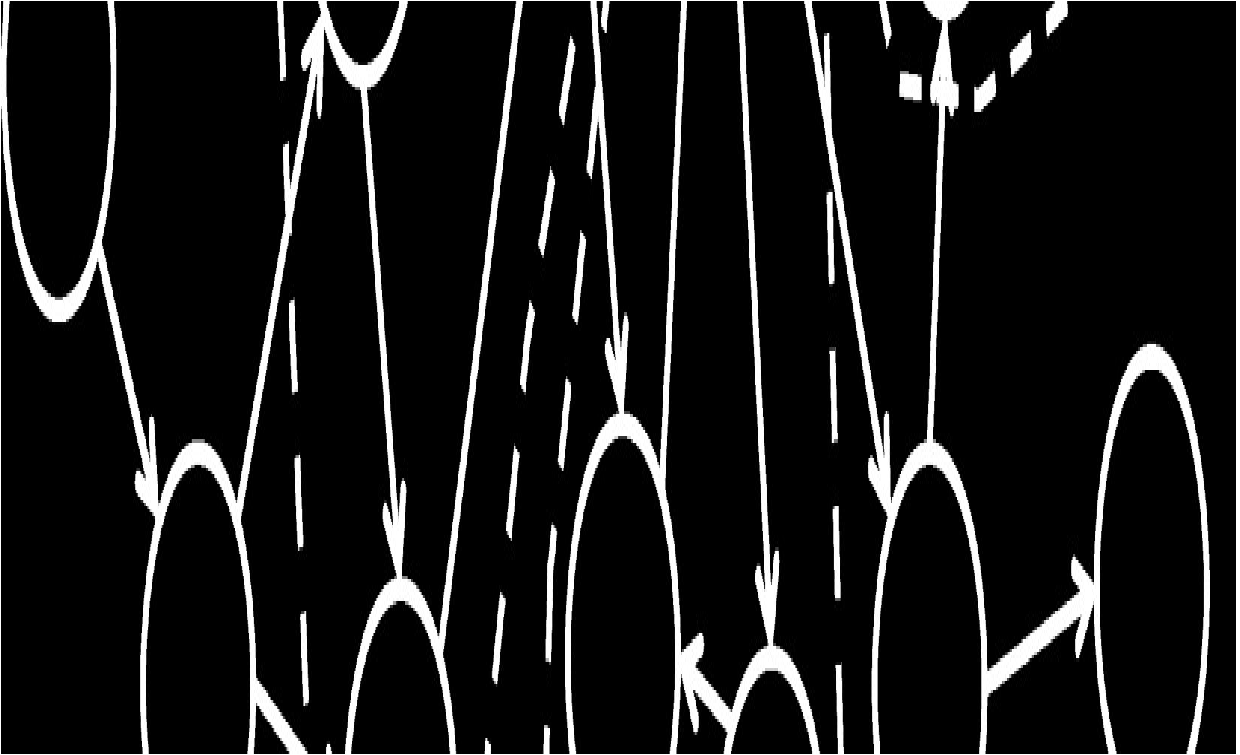
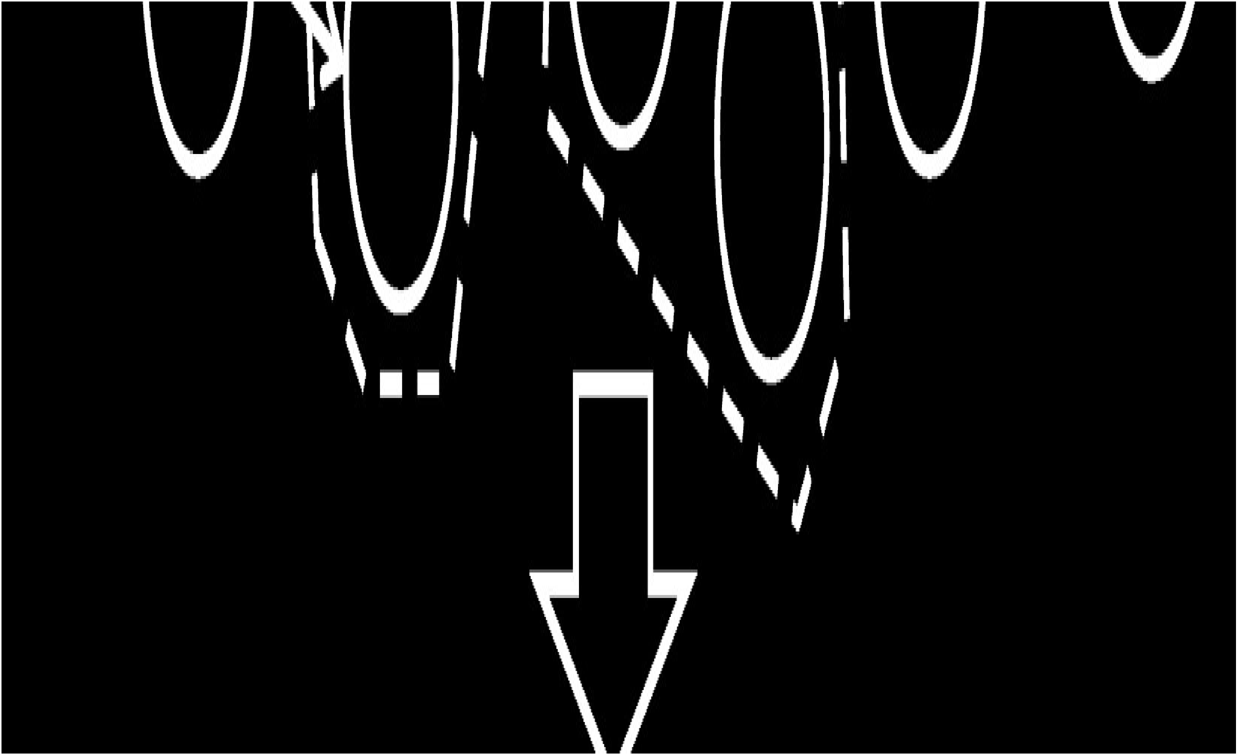

```
// 求区域的个数
int ans = 0;
for (int y = 0; y < H; y++) {
for (int x = 0; x < W; x++) {
if (fld[y][x]) continue;
ans++;
// 宽度优先搜索
queue<pair<int, int> > que;
que.push(make_pair(x, y));
while (!que.empty()) {
int sx = que.front().first, sy = que.front().second;
que.pop();
for (int i = 0; i < 4; i++) {
int tx = sx + dx[i], ty = sy + dy[i];
if (tx < 0 || W <= tx || ty < 0 || H <= ty) continue; if (fld[ty][tx]) continue;
que.push(make_pair(tx, ty));
fld[ty][tx] = true;
}
```

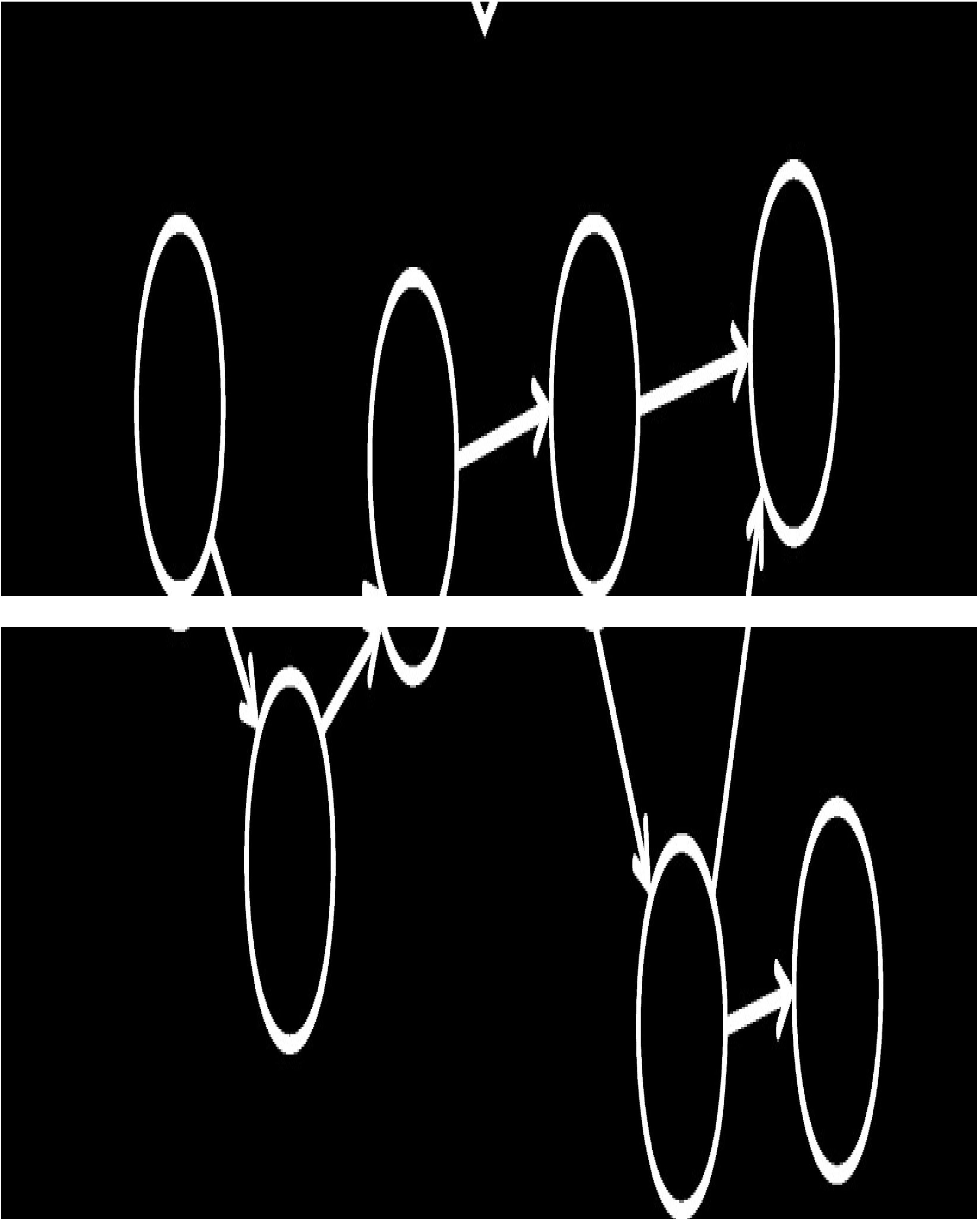
```
}
}
}
printf("%d\n", ans);
}
```

第 4 章

登峰造极——高级篇





图是非常有用的数据结构，除了在第2章已经介绍的算法外，还有各种各样的相关算法。在此，

我们主要讨论强连通分量分解和最近公共祖先等问题。

4.3.1 强连通分量分解

对于一个有向图顶点的子集 S ，如果在 S 内任取两个顶点 u 和 v ，都能找到一条从 u 到 v 的路径，那么

就称 S 是强连通的。如果在强连通的顶点集合 S 中加入其他任意顶点集合后，它都不再是强连通的，

那么就称 S 是原图的一个强连通分量（SCC: Strongly Connected Component）。任意有向图都可以

分解成若干不相交的强连通分量，这就是强连通分量分解。把分解后的强连通分量缩成一个顶点，

就得到了一个DAG（有向无环图）。

虚线包围的部分构成一个强连通分量

强连通分量分解可以通过两次简单的DFS实现。第一次DFS时，选取任意顶点作为起点，遍历所

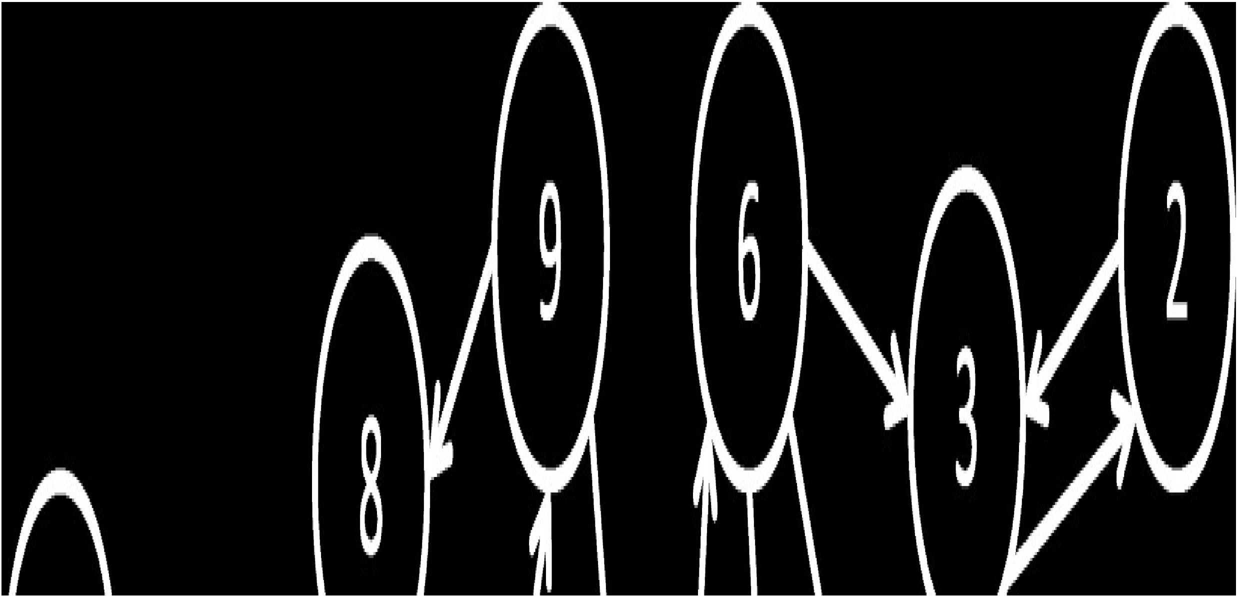
有尚未访问过的顶点，并在回溯前给顶点标号（post order，后序遍历）。对剩余的未访问过的顶

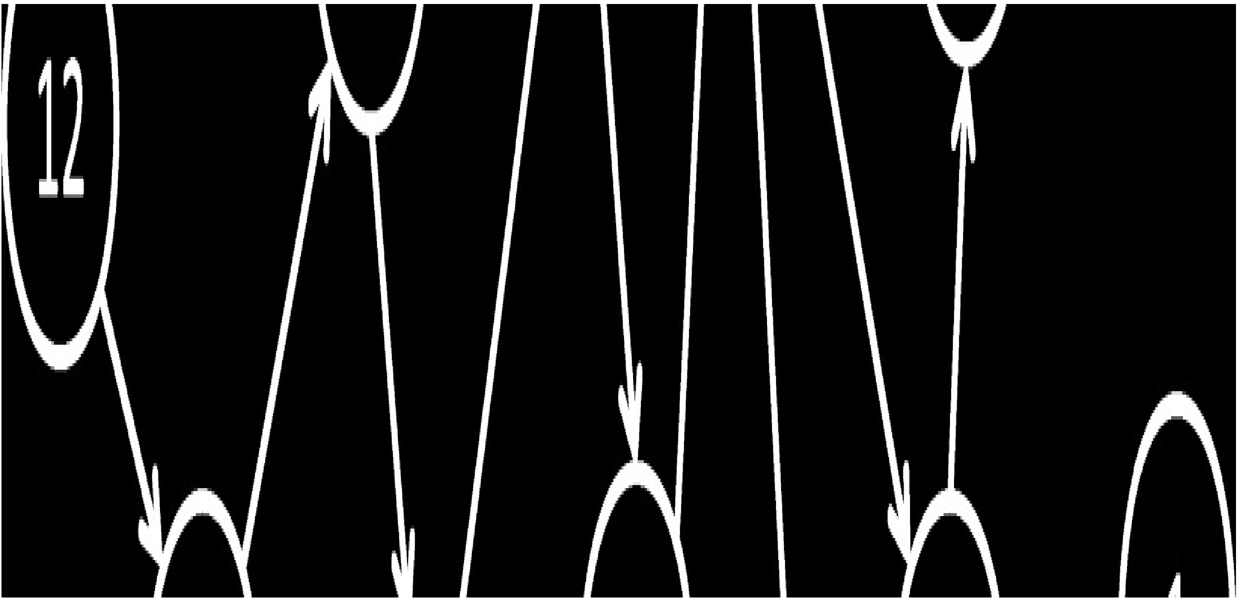
点，不断重复上述过程。

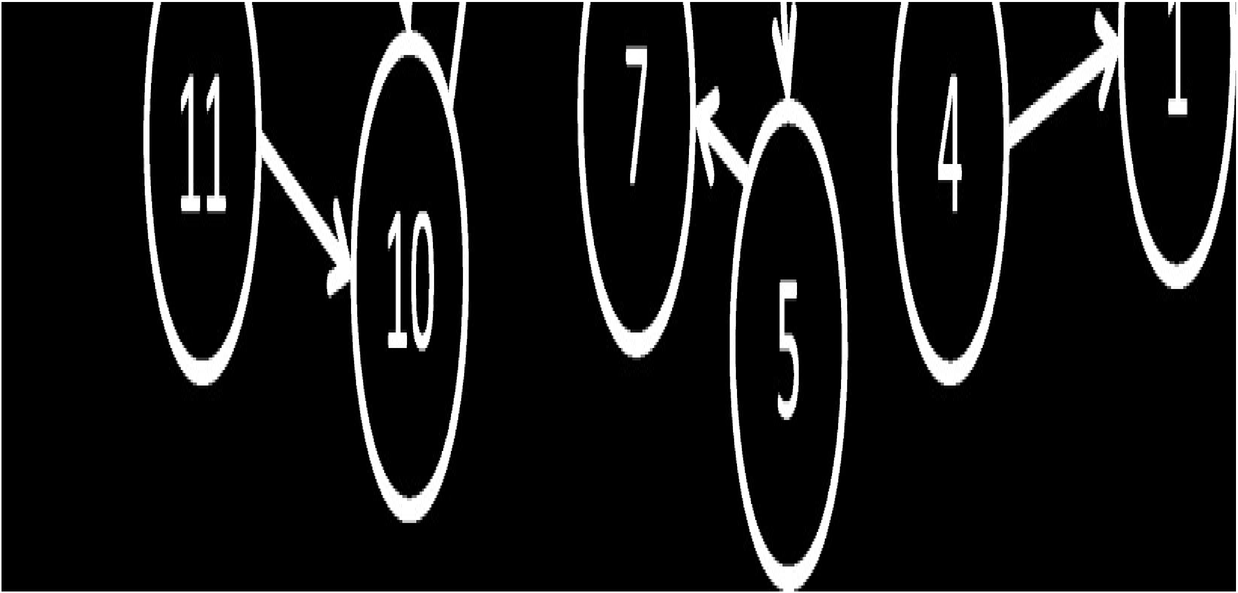
完成标号后，越接近图的尾部（搜索树的叶子），顶点的标号越小。第二次DFS时，先将所有边

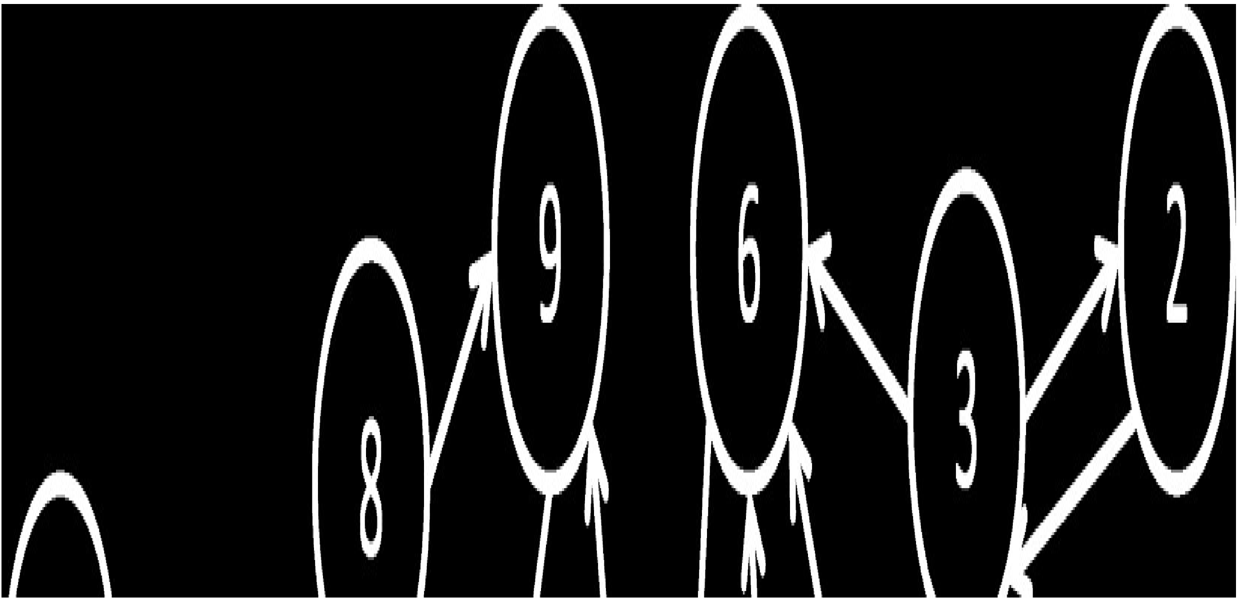
反向，然后以标号最大的顶点为起点进行DFS。这样DFS所遍历的顶点集合就构成了一个强连通

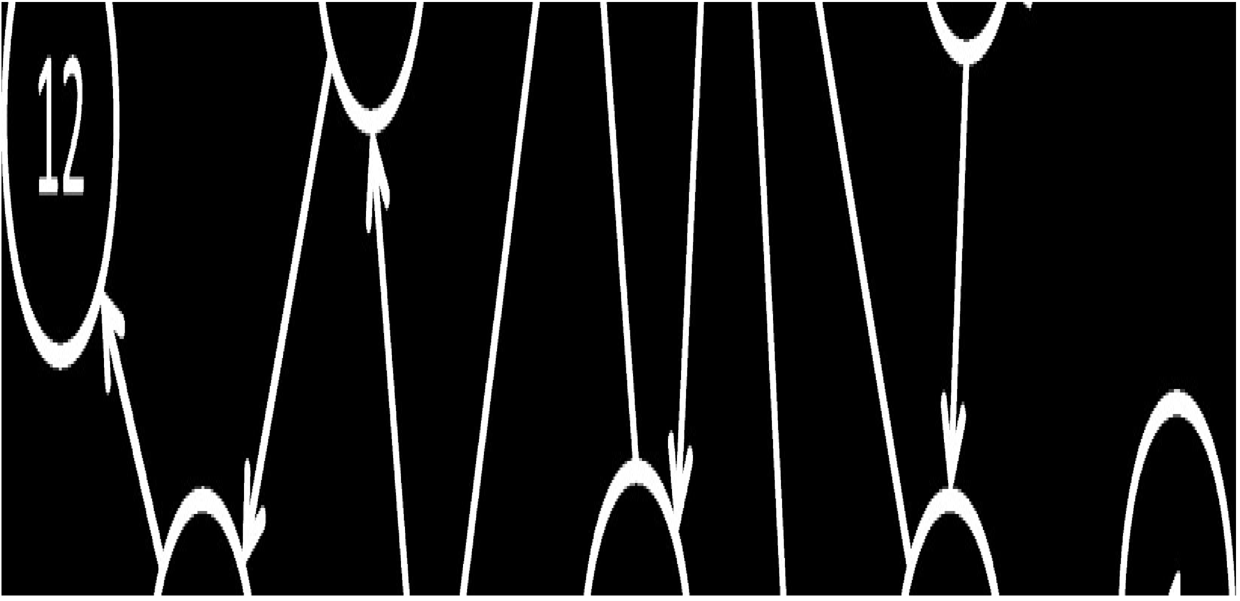
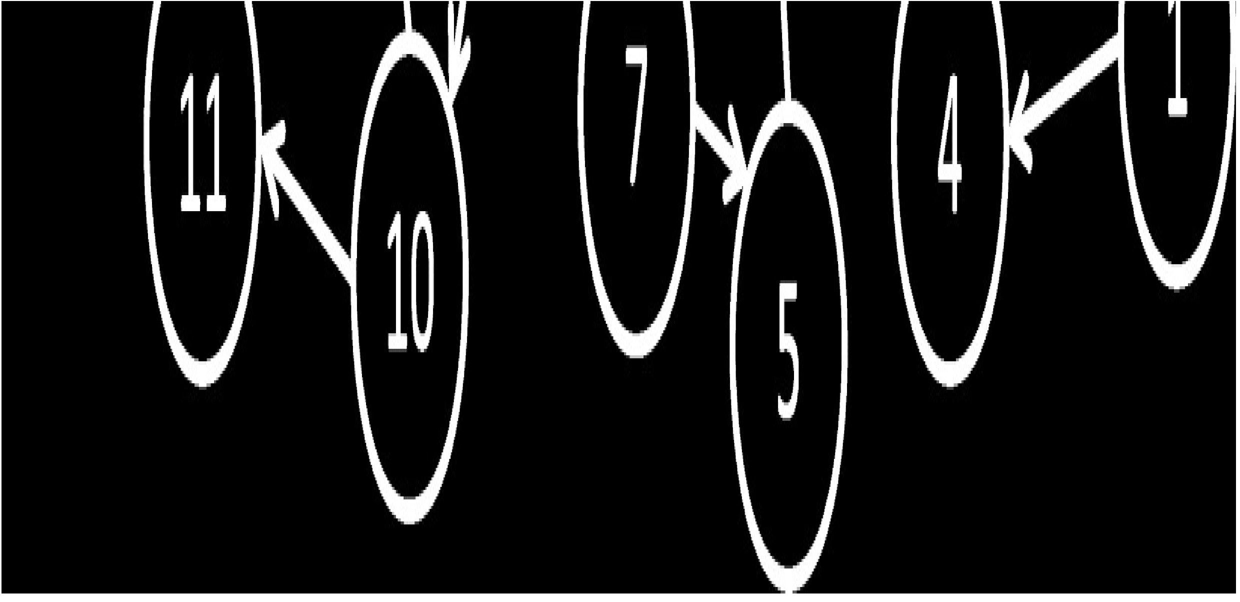
分量。之后，只要还有尚未访问的顶点，就从中选取标号最大的顶点不断重复上述过程。

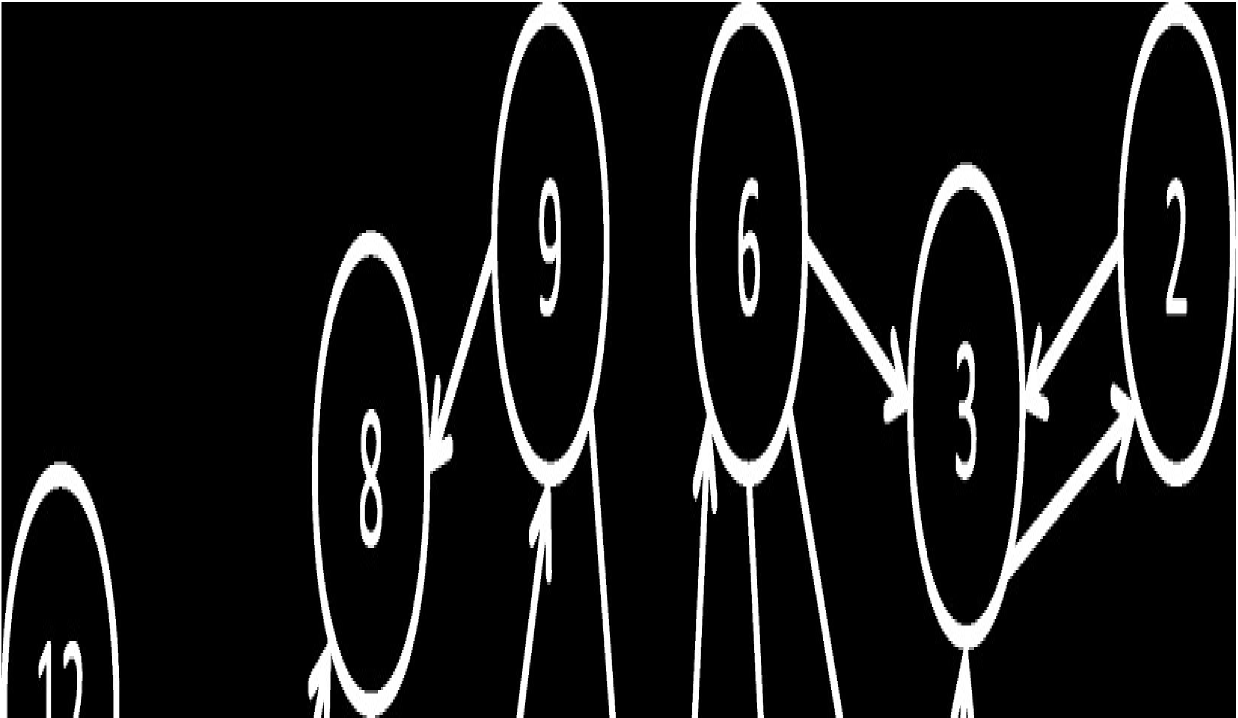

9
6
2
8
3

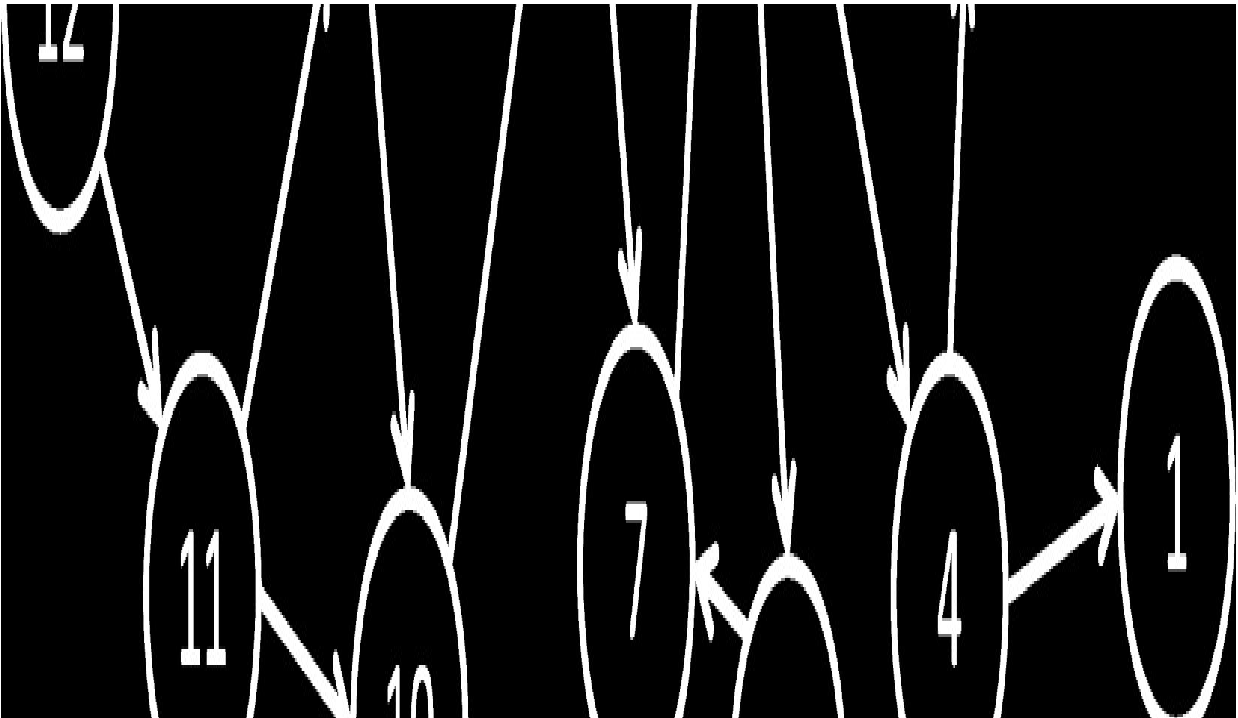

12

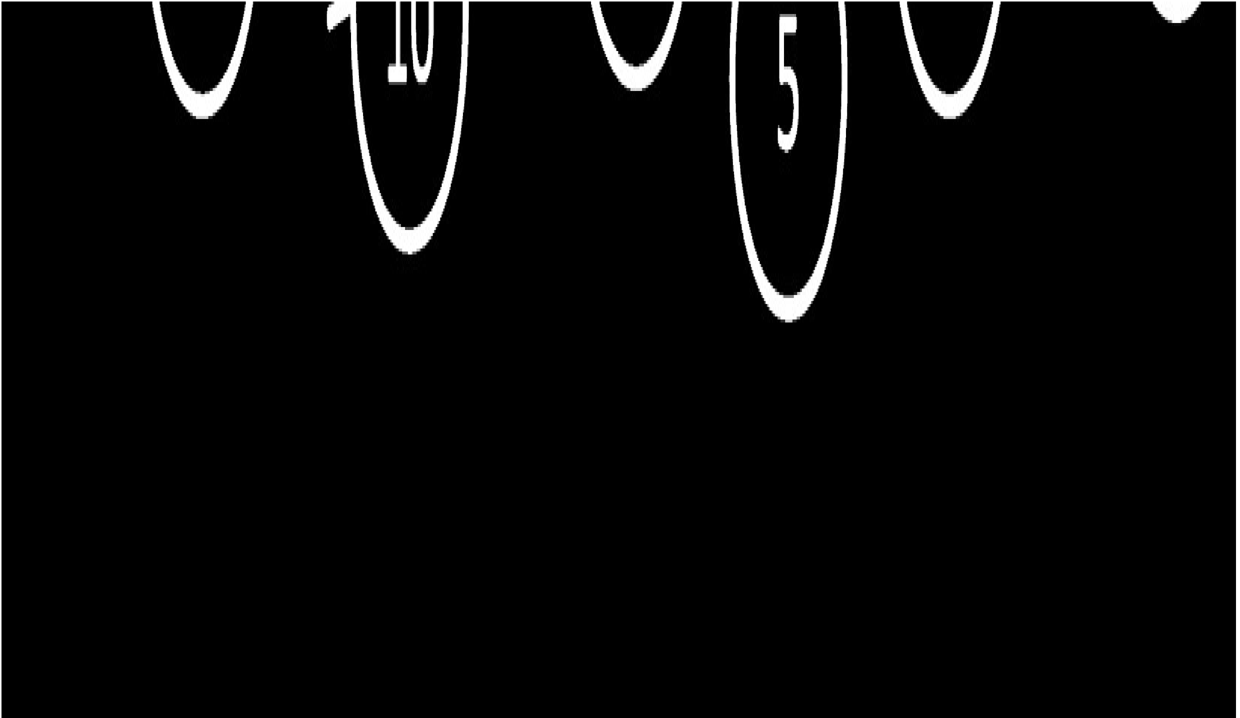

11
10
7
5
4
1

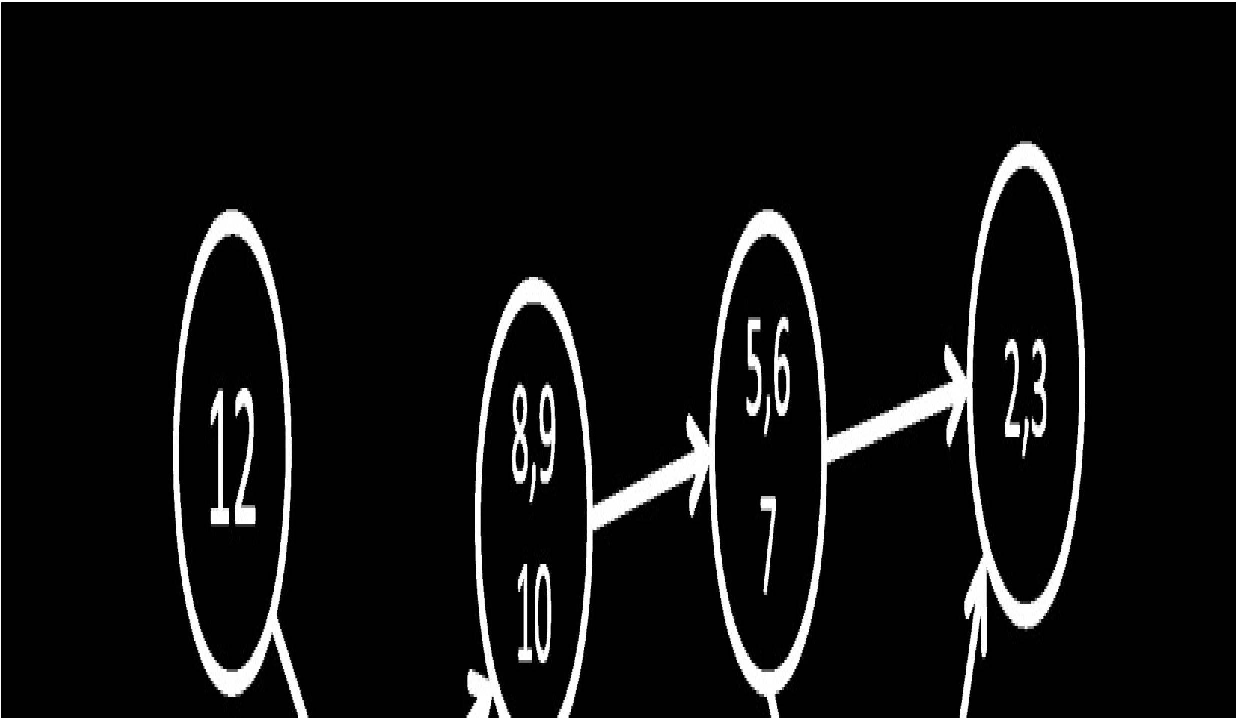

9
6
3
2
8

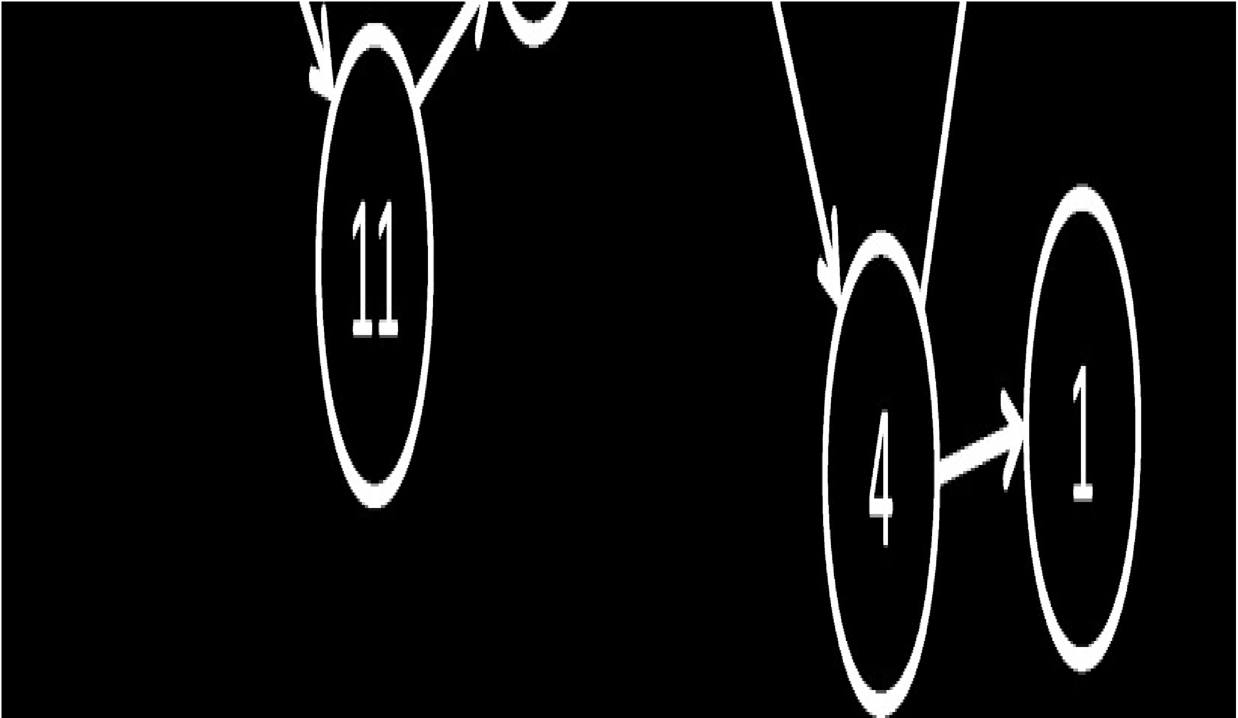

12

11
10
7
5
4
1


9
6
2
3
8
12


12
11
7
4
1


10
5


12
8,9
10
5,6
7
2,3



1

2

后续遍历的例子。根据搜索顺序的不同，标号结果也可能不同

3

4

反向后的图

正如前文所述，我们可以将强连通分量缩点并得到DAG。此时可以发现，标号最大的节点就属于

5

DAG头部（搜索树的根）的强连通分量。因此，将边反向后，就不能沿边访问到这个强连通分量

以外的顶点。而对于强连通分量内的其他顶点，其可达性不受边反向的影响，因此在第二次DFS

时，我们可以遍历一个强连通分量里的所有顶点。

6

7

8

9

边反向后，从8、9、10号顶点只能到达其头部方向的顶点11和12

该算法只进行了两次DFS，因而总的复杂度是 $O(|V|+|E|)$。

int V; // 顶点数

10

vector<int> G[MAX_V]; // 图的邻接表表示

vector<int> rG[MAX_V]; // 把边反向后的图

vector<int> vs; // 后序遍历顺序的顶点列表

bool used[MAX_V]; // 访问标记

int cmp[MAX_V]; // 所属强连通分量的拓扑序

11



```
void add_edge(int from, int to) {
G[from].push_back(to);
rG[to].push_back(from);
}
void dfs(int v) {
used[v] = true;
for (int i = 0; i < G[v].size(); i++) {
if (!used[G[v][i]]) dfs(G[v][i]);
}
vs.push_back(v);
}
void rdfs(int v, int k) {
used[v] = true;
cmp[v] = k;
for (int i = 0; i < rG[v].size(); i++) {
if (!used[rG[v][i]]) rdfs(rG[v][i], k);
}
}
int scc() {
```

```
memset(used, 0, sizeof(used));

vs.clear();

for (int v = 0; v < V; v++) {

if (!used[v]) dfs(v);

}

memset(used, 0, sizeof(used));

int k = 0;

for (int i = vs.size() - 1; i >= 0; i--) {

if (!used[vs[i]]) rdfs(vs[i], k++);

}

return k;

}
```

Popular Cows （POJ No.2186）

每头牛都想成为牛群中的红人。给定 N 头牛的牛群和 M 个有序对(A , B)。(A , B)表示牛 A 认

为牛 B 是红人。该关系具有传递性，所以如果牛 A 认为牛 B 是红人，牛 B 认为牛 C 是红人，

那么牛 A 也认为牛 C 是红人。不过，给定的有序对中可能包含(A , B)和(B , C)，但不包含(A , C)。

求被其他所有牛认为是红人的牛的总数。

限制条件

 $1 \leq N \leq 10000$

$1 \le M \le 50000$

$1 \le A, B \le N$

样例

1

输入

N = 3

M = 3

(A, B) = {(1, 2), (2, 1), (2, 3)}

2

输出

1（3号牛）

3

考虑以牛为顶点的有向图，对每个有序对(A, B)连一条从A到B的有向边。那么，被其他所有牛认

为是红人的牛对应的顶点，也就是从其他所有顶点都可达的顶点。虽然这可以通过从每个顶点出

发搜索求得，但总的复杂度却是$O(NM)$，是不可行的，必须要考虑更为高效的算法。

4

假设有两头牛A和B都被其他所有牛认为是红人。那么显然，A被B认为是红人，B也被A认为是红

人，即存在一个包含A、B两个顶点的圈，或者说，A、B同属于一个强连通分量。反之，如果一

头牛被其他所有牛认为是红人，那么其所属的强连通分量内的所有牛都被其他所有牛认为是红

5

人。由此，我们把图进行强连通分量分解后，至多有一个强连通分量满足题目的条件。而按前面

介绍的算法进行强连通分量分解时，我们还能够得到各个强连通分量拓扑排序后的顺序，唯一可

能成为解的只有拓扑序最后的强连通分量。所以在最后，我们只要检查这个强连通分量是否从所

6

有顶点可达就好了。该算法的复杂度为 $O(N+M)$，足以在时限内解决原题。

```
// 输入

int N, M;

int A[MAX_M], B[MAX_M];
```

7

```
void solve() {

V = N;

for (int i = 0; i < M; i++) {

add_edge(A[i] - 1, B[i] - 1);
```

8

```
}
int n = scc();
// 统计备选解的个数
int u = 0, num = 0;
```

9

```
for (int v = 0; v < V; v++) {
if (cmp[v] == n - 1) {
u = v;
num++;
}
```

10

```
}
// 检查是否从所有点可达
memset(used, 0, sizeof(used));
rdfs(u, 0); // 重用强连通分量分解的代码
```

11


```
for (int v = 0; v < V; v++) {
if (!used[v]) {
// 从该点不可达
```

```
num = 0;

break;

}

}

printf("%d\n", num);

}
```

4.3.2 2-SAT

给定一个布尔方程， 判断是否存在一组布尔变量的真值指派使整个方程为真的问题，被称为布

尔方程的可满足性问题（SAT）。SAT问题是NP完全的，但对于满足一定限制条件的SAT问题，还

是能够有效求解的。我们将下面这种布尔方程称为合取范式。

$(a \vee b \vee \ldots) \wedge (c \vee d \vee \ldots) \wedge \ldots$

其中 $a, b, \ldots$称为文字，它是一个布尔变量或其否定。像$(a \vee b \vee ...)$这样用∨连接的部分称为子句。

如果合取范式的每个子句中的文字个数都不超过两个，那么对应的SAT问题又称为2-SAT问题。

□ 2-SAT布尔公式的例子

□ $(a \vee b) \wedge \neg a$ 令 a 为假而 b 为真，则可以满足

□ $(a \vee \neg b) \wedge (b \vee c) \wedge (\neg c \vee \neg a)$ 令 a 和 b 为真而 c 为假，则可以满足

□ $(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$ 无法满足

利用强连通分量分解，可以在布尔公式子句数的线性时间内解决2-SAT问题。首先，利用⇒（蕴

涵）将每个子句($a \vee b$)改写成等价形式($\neg a \Rightarrow b \wedge \neg b \Rightarrow a$)。这样原布尔公式就变成了把 $a \Rightarrow b$ 形式的

布尔公式用∧连接起来的形式。对每个布尔变量 x，构造两个顶点分别代表 x 和 $\neg x$，以⇒关系为边

建立有向图。此时，如果图上的 a 点能够到达 b 点的话，就表示当 a 为真时 b 也一定为真。因此，该

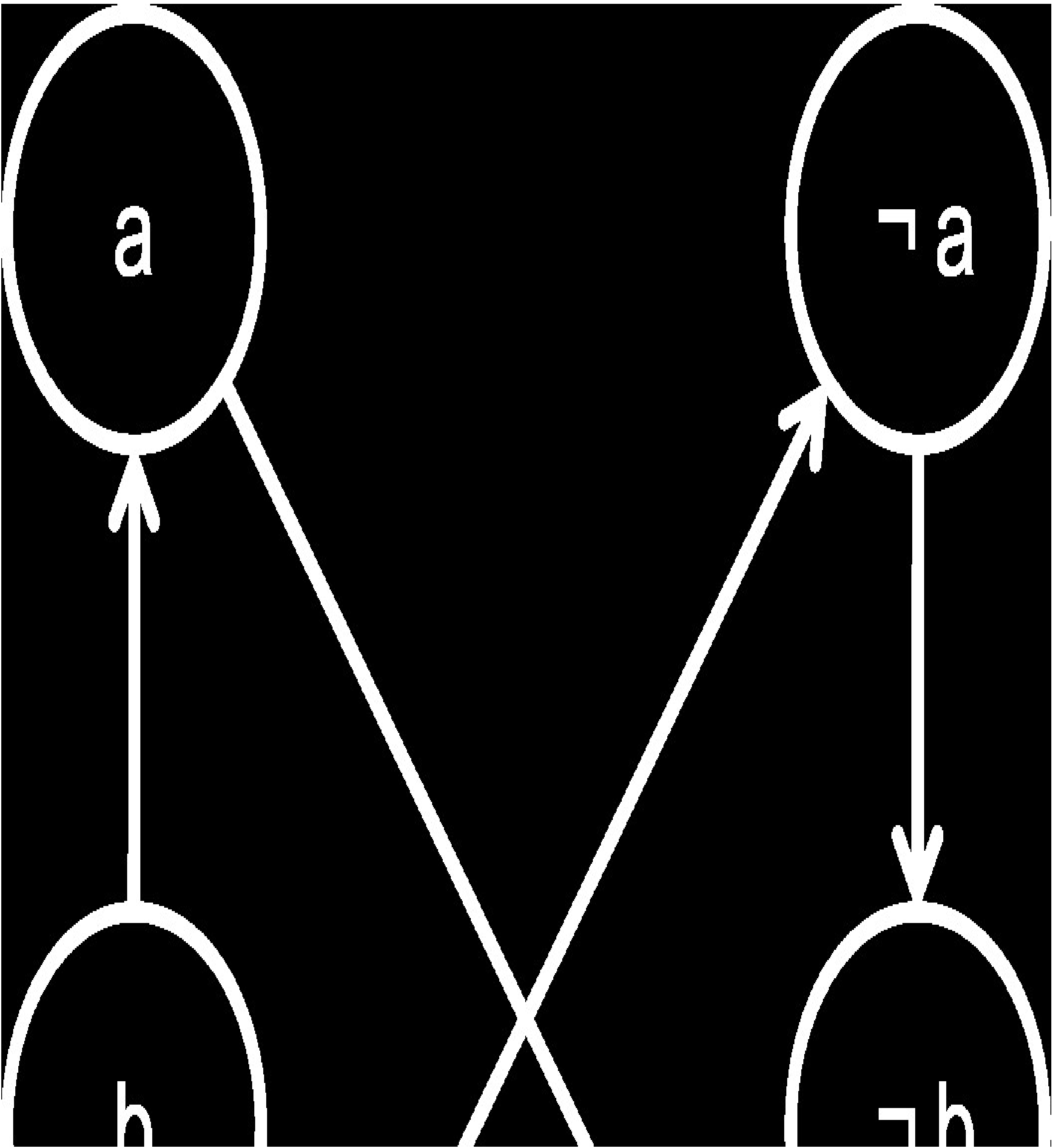
图中同一个强连通分量中所含的所有文字的布尔值均相同。

如果存在某个布尔变量 x，x 和 $\neg x$ 均在同一个强连通分量中，则显然无法令整个布尔公式的值为真。

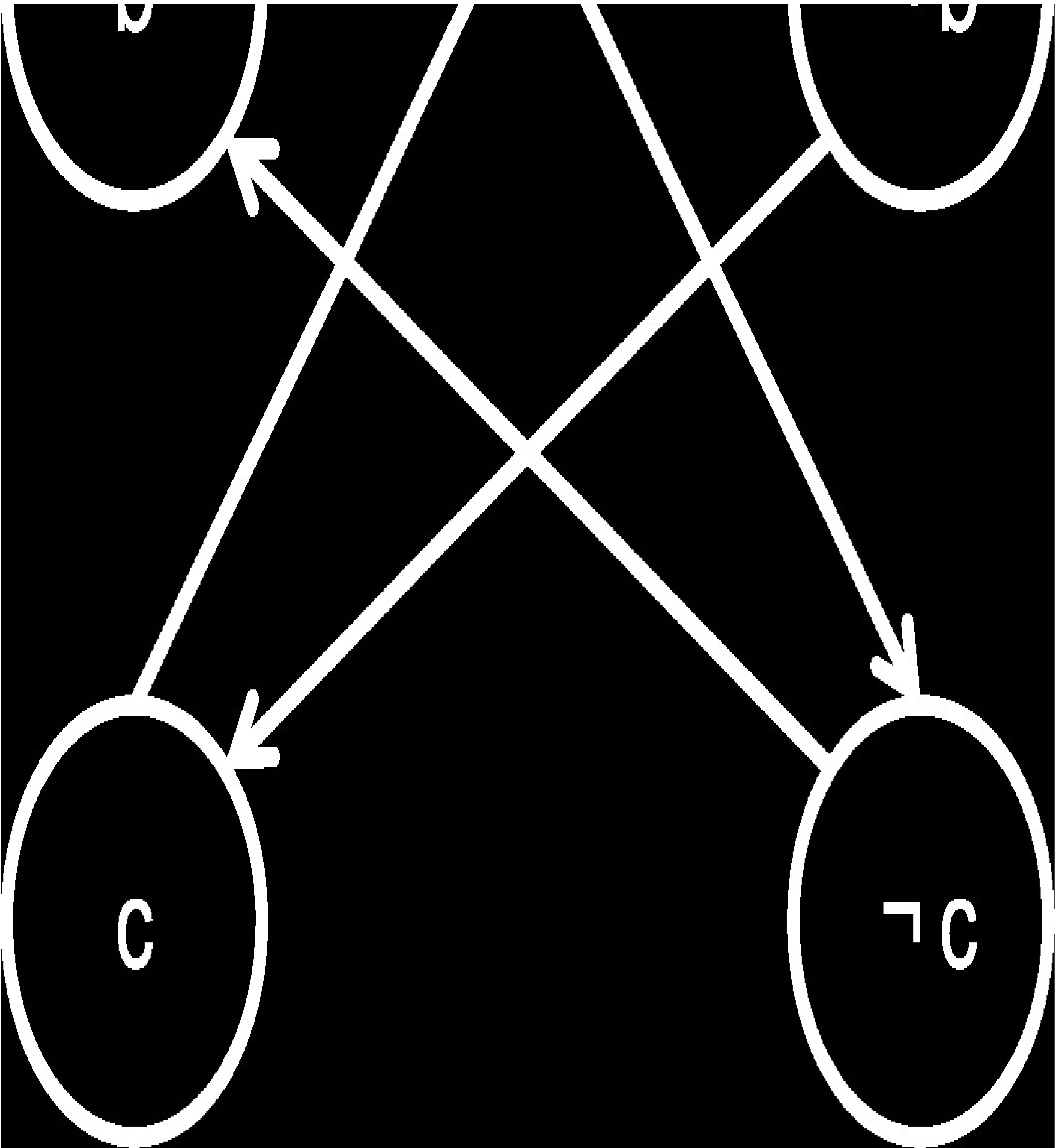
反之，如果不存在这样的布尔变量，那么对于每个布尔变量 x，让

x 所在的强连通分量的拓扑序在 $\neg x$ 所在的强连通分量之后 $\Leftrightarrow$ x 为真

就是使得该公式的值为真的一组合适的布尔变量赋值。



a
¬a



1

2

3

(a∨¬b)∧(b∨c)∧(¬c∨¬a)所对应的图

int main() {

// 布尔公式为(a∨ ¬b)∧ (b∨ c)∧ (¬c∨ ¬a)时

// 构造6个顶点，分别对应a、b、c、¬a、¬b、¬c。

4

V = 6;

// a∨ ¬b转成¬a⇒¬b∧b⇒a

add_edge(3, 4); // 从¬a连一条到¬b的边

add_edge(1, 0); // 从b连一条到a的边

5

// b∨ c转成¬b⇒c∧¬c⇒b

add_edge(4, 2); // 从¬b连一条到c的边

add_edge(5, 1); // 从¬c连一条到b的边

// ¬c∨ ¬a转成c⇒¬a∧a⇒¬c

add_edge(2, 3); // 从c连一条到¬a的边

6

add_edge(0, 5); // 从a连一条到¬c的边

// 进行强连通分量分解

scc();

// 判断是否x和¬x在不同的强连通分量中

7

```
for (int i = 0; i < 3; i++) {
if (cmp[i] == cmp[3 + i]) {
printf("NO");
return 0;
}
```

8

```
}
// 如果可满足，则给出一组解
printf("YES\n");
for (int i = 0; i < 3; i++) {
```

9

```
if (cmp[i] > cmp[3 + i]) {
printf("true\n");
} else {
printf("false\n");
}
```

10

```
}
return 0;
```

}

11



Priest John’s Busiest Day（POJ No.3683）

约翰是街区里唯一的神父。假设有 N 对新人打算在同一天举行结婚仪式。第 i 对新人的结婚

仪式的时间为 S_i 到 T_i，在其仪式开始时或是结束时需要进行一个用时 D_i 的特别仪式（也就

是从 S_i 到 $S_i + D_i$ 或是从 $T_i - D_i$ 到 T_i），该特别仪式需要神父在场。请判断是否可以通过合理安

排每个特别仪式在开始还是结束时进行，从而保证神父能够出席所有的特别仪式。如果可能

的话，请输出出席各个特别仪式的时间。当然，神父不可能同时出席多个特别仪式。不过神

父前往仪式的途中所花费的时间可以忽略不计，神父可以在出席完一个特别仪后，立刻出席

另一个开始时间与其结束时间相等的特别仪式。

限制条件

- $1 \le N \le 1000$

样例

输入

N = 2

(S, T, D) = {(08:00, 09:00, 30), (08:15, 09:00, 20)}

输出

YES

08:00 08:30

08:40 09:00

对于每个结婚仪式 i，只有在开始或结束时进行特别仪式两种选择。因此可以定义变量 x_i

x_i 为真 ⇔ 在开始时进行特别仪式

这样，对于结婚仪式 i 和 j，如果 $S_i \sim S_i + D_i$ 和 $S_j \sim S_j + D_j$ 冲突，就有 $\neg x_i \vee \neg x_j$ 为真。对于开始和结束、结

束和开始、结束和结束等三种情况，也可以得到类似的条件。于是，要保证所有特别仪式的时间

不冲突，只要考虑将这所有的子句用∧连接起来所得到的布尔公式就好了。例如，对于输入样例，

可以的到布尔公式

$(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ 而当 x_1为真而 x_2为假时，其值为真。这样，我们就把原问题转为了2-SAT问题。接下来只要进行强

连通分量分解并判断是否有使得布尔公式值为真的一组布尔变量赋值就好了。

// 输入

```
int N;
int S[MAX_N], T[MAX_N], D[MAX_N]; // S和T是换算成分钟后的时间
```


1

```
void solve() {
// 0~N-1：x_i
// N~2N-1: ¬x_i
V = N * 2;
for (int i = 0; i < N; i++) {
for (int j = 0; j < i; j++) {
```

2

```
if (min(S[i] + D[i], S[j] + D[j]) > max(S[i], S[j])) {
// x_i⇒¬x_j 、x_j⇒¬x_i
add_edge(i, N + j);
add_edge(j, N + i);
}
```

3

```
if (min(S[i] + D[i], T[j]) > max(S[i], T[j] - D[j])) {
// x_i⇒x_j 、¬x_j⇒¬x_i
add_edge(i, j);
```

add_edge(N + j, N + i);

}

4

if (min(T[i], S[j] + D[j]) > max(T[i] - D[i], S[j])) {

// ¬x_i⇒¬x_j、x_j⇒x_i

add_edge(N + i, N + j);

add_edge(j, i);

}

5

if (min(T[i], T[j]) > max(T[i] - D[i], T[j] - D[j])) {

// ¬x_i⇒x_j、¬x_j⇒x_i

add_edge(N + i, j);

add_edge(N + j, i);

}

6

}

}

scc();

// 判断是否可满足

7

```
for (int i = 0; i < N; i++) {

if (cmp[i] == cmp[N + i]) {

printf("NO\n");

return;

}

}
```

8

```
// 如果可满足，则给出一组解

printf("YES\n");

for (int i = 0; i < N; i++) {

if (cmp[i] > cmp[N + i]) {
```

9

```
// x_i为真，即在结婚仪式开始时举行

printf("%02d:%02d %02d:%02d\n", S[i] / 60, S[i] % 60, (S[i] + D[i]) / 60,
(S[i] + D[i]) % 60);

} else {

// x_i为假，即在结婚仪式结束时举行
```

10

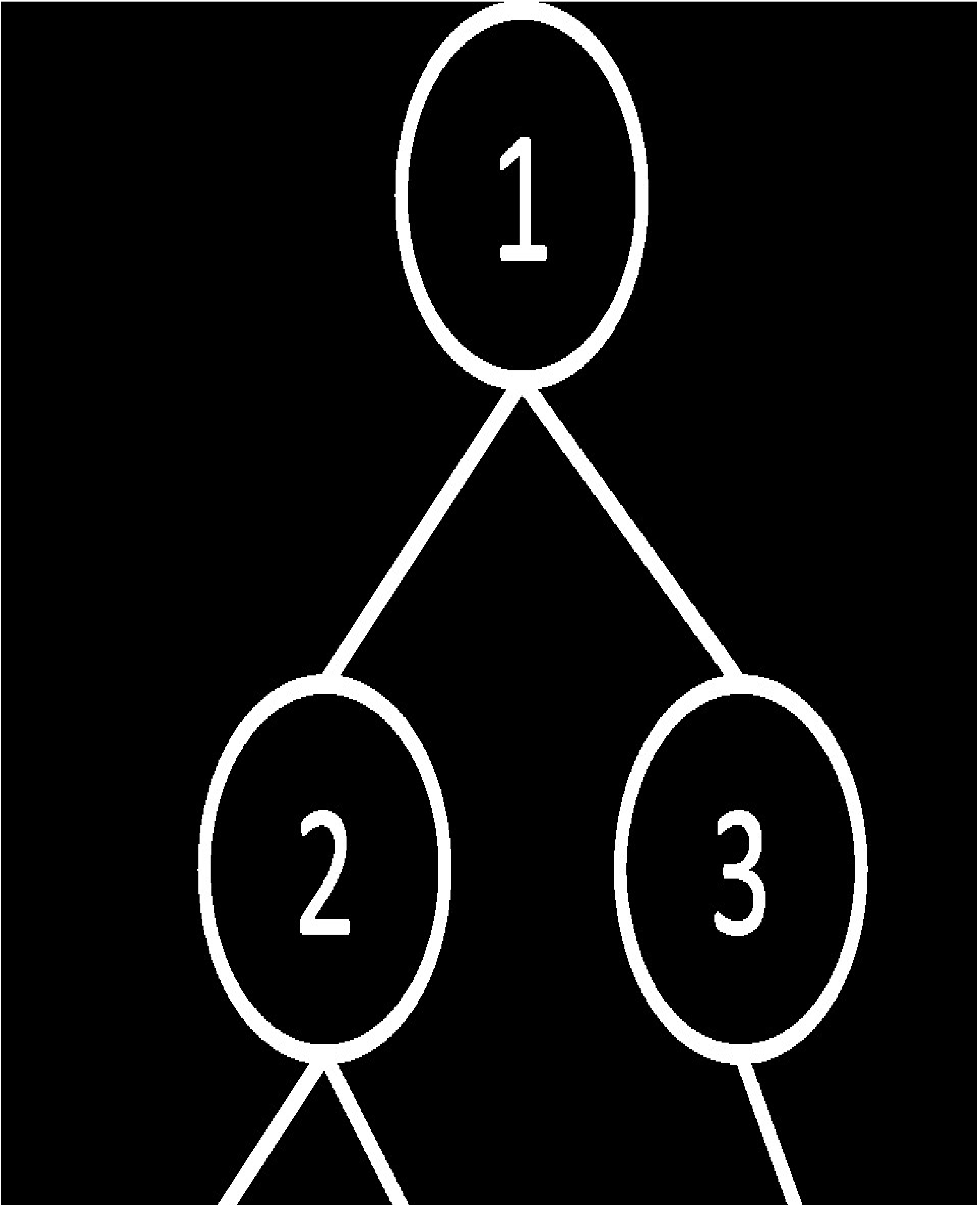
```
printf("%02d:%02d %02d:%02d\n", (T[i] - D[i]) / 60, (T[i] - D[i]) % 60,
T[i] / 60, T[i] % 60);

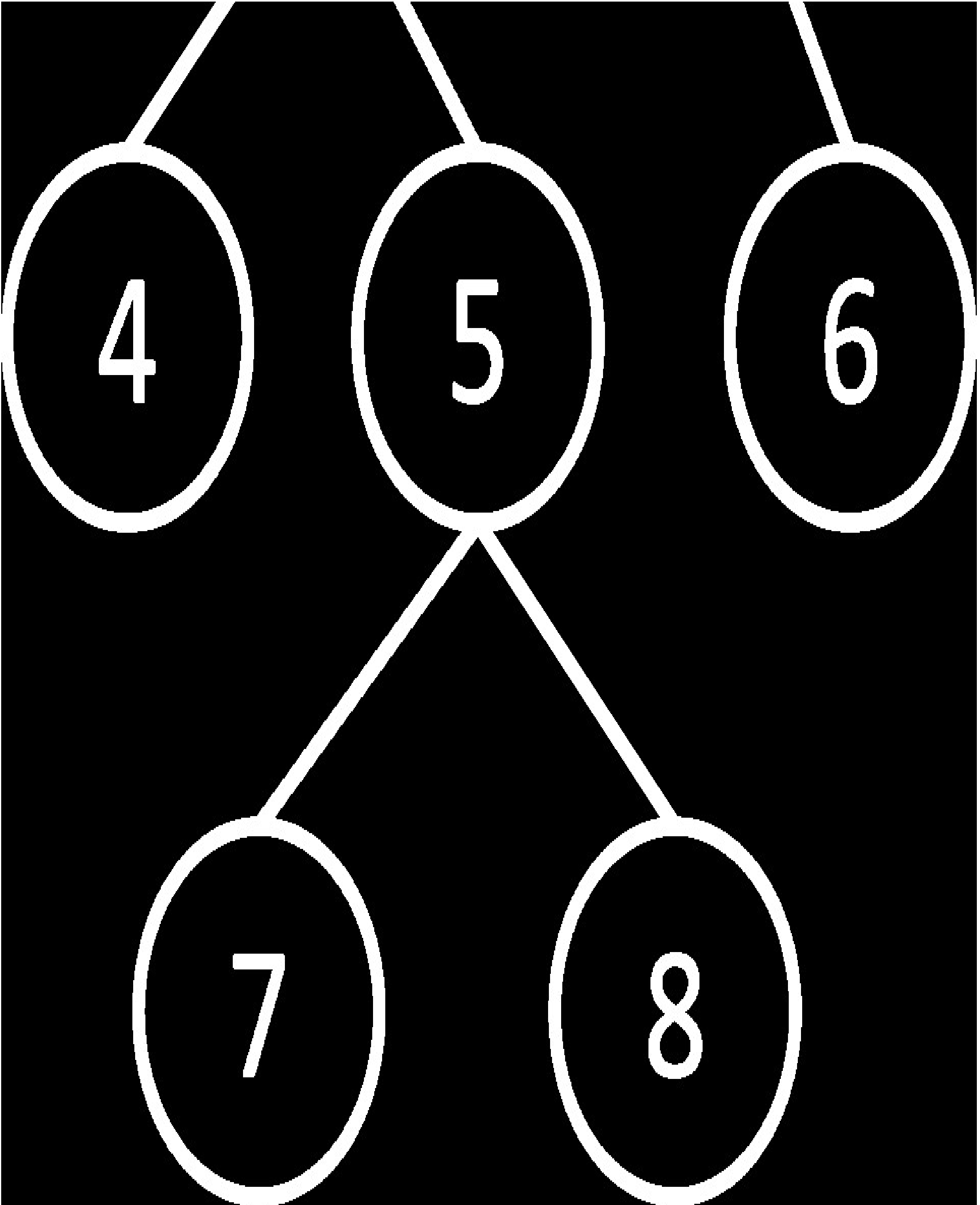
}
```

}

}

11


1
2
3



4.3.3 LCA

在有根树中，两个节点 u 和 v 的公共祖先中距离最近的那个被称为最近公共祖先（LCA，Lowest Common Ancestor)。用于高效计算LCA的算法有许多，在此我们介绍其中的两种。在下文中，我

们都假设节点数为 n 。

LCA的例子（4和7的LCA为2，8和6的LCA为1，5和8的LCA为5）

1. 基于二分搜索的算法

记节点 v 到根的深度为depth(v) 。那么，如果节点 w 是 u 和 v 的公共祖先的话，让 u 向上走

depth(u)-depth(w)步，让 v 向上走depth(v)-depth(w)步，就都将走到 w 。因此，首先让 u 和 v 中较深的

一方向上走|depth(u)-depth(v)|步，再一起一步步向上走，直到走到同一个节点，就可以在

O (depth(u)+depth(v))时间内求出LCA。

```
// 输入

vector<int> G[MAX_V]; // 图的邻接表表示

int root; // 根节点的编号

int parent[MAX_V]; // 父亲节点（根节点的父亲记为-1）

int depth[MAX_V]; // 节点的深度

void dfs(int v, int p, int d) {

parent[v] = p;

depth[v] = d;

for (int i = 0; i < G[v].size(); i++) {

if (G[v][i] != p) dfs(G[v][i], v, d + 1);
```

```
}
}
// 预处理
void init() {
// 预处理出parent和depth
dfs(root, -1, 0);
}
// 计算u和v的LCA
```



```
int lca(int u, int v) {
```

1

```
// 让u和v向上走到同一深度
while (depth[u] > depth[v]) u = parent[u];
while (depth[v] > depth[u]) v = parent[v];
// 让u和v向上走到同一节点
while (u != v) {
u = parent[u];
```

2

```
v = parent[v];
}
```

return u;

}

3

节点的最大深度是 $O(n)$，所以该算法的复杂度也是 $O(n)$。如果只需计算一次LCA的话，这便足够

了。但如果要计算多对节点的LCA的话如何是好呢？刚才的算法，通过不断向上走到同一节点来

计算 u 和 v 的LCA。这里，到达了同一节点后，不论再怎么向上走，到达的显然还是同一节点。利

4

用这一点，我们能够利用二分搜索求出到达共同祖先所需的最少步数吗？事实上，只要利用如下

预处理，就可以实现二分搜索。

首先，对于任意顶点 v，利用其父亲节点信息，可以通过parent2[v]=parent[parent[v]]得到其向上走

5

两步所到的顶点。再利用这一信息，又可以通过parent4[v]=parent2[parent2[v]]得到其向上走四步

所到的顶点。依此类推，就能够得到其向上走2 k 步所到的顶点parent[k][v]。有了 k =floor(log n)以内

的所有信息后，就可以二分搜索了，每次的复杂度是 $O(\log n)$。另外，预处理parent[k][v]的复杂度

6

是 $O(n \log n)$。

```
// 输入
vector<int> G[MAX_V]; // 图的邻接表表示
int root; // 根节点的编号
```

7

```
int parent[MAX_LOG_V][MAX_V]; // 向上走2^k步所到的节点（超过根时记为-1）
int depth[MAX_V]; // 节点的深度
void dfs(int v, int p, int d) {
```

8

```
parent[0][v] = p;
depth[v] = d;
for (int i = 0; i < G[v].size(); i++) {
if (G[v][i] != p) dfs(G[v][i], v, d + 1);
}
}
```

9

```
// 预处理
void init(int V) {
// 预处理出parent[0]和depth
dfs(root, -1, 0);
```

10

```
// 预处理出parent
for (int k = 0; k + 1 < MAX_LOG_V; k++) {
for (int v = 0; v < V; v++) {
if (parent[k][v] < 0) parent[k + 1][v] = -1;
else parent[k + 1][v] = parent[k][parent[k][v]];
```

11

```
}
```

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vs	1	2	4	2	5	7	5	8	5	2	1	3	6	3	1
depth	0	1	2	1	2	3	2	3	2	1	0	1	2	1	0

i	1	2	3	4	5	6	7	8
id	0	1	11	2	4	12	5	7


```
}
}
// 计算u和v的LCA
int lca(int u, int v) {
// 让u和v向上走到同一深度
if (depth[u] > depth[v]) swap(u, v);
for (int k = 0; k < MAX_LOG_V; k++) {
if ((depth[v] - depth[u]) >> k & 1) {
v = parent[k][v];
}
}
if (u == v) return u;
// 利用二分搜索计算LCA
```

```
for (int k = MAX_LOG_V - 1; k >= 0; k--) {

if (parent[k][u] != parent[k][v]) {

u = parent[k][u];

v = parent[k][v];

}

}

return parent[0][u];

}
```

像这样，预处理出2 k 的表的技巧，在计算LCA之外也很有用，相关的问题也经常出现在程序设计

竞赛当中。对此，下一节中还会介绍其他例子。

2. 基于RMQ的算法

对于涉及有根树的问题，将树转为从根DFS标号后得到的序列处理的技巧常常十分有效。对于

LCA，利用该技巧也能够高效地计算。首先，按从根DFS访问的顺序得到顶点序列vs[i]和对应的

深度depth[i]。对于每个顶点 v ，记其在 vs 中首次出现的下标为id[v]。

样例对应的标号

这些都可以在 $O(n)$时间内求得。而LCA(u , v)就是访问 u 之后到访问 v 之前所经过顶点中离根最近的

那个，假设id[u]≤id[v]，那么有

$LCA(u,v)= vs[id[u]\le i \le id[v]$中令$depth(i)$最小的$i]$

而这可以利用RMQ高效地求得。



// 输入

1

```
vector<int> G[MAX_V]; // 图的邻接表表示
int root;
int vs[MAX_V * 2 - 1]; // DFS访问的顺序
int depth[MAX_V * 2 - 1]; // 节点的深度
```

2

```
int id[MAX_V]; // 各个顶点在vs中首次出现的下标
void dfs(int v, int p, int d, int &k) {
id[v] = k;
vs[k] = v;
```

3

```
depth[k++] = d;
for (int i = 0; i < G[v].size(); i++) {
```

```
if (G[v][i] != p) {
dfs(G[v][i], v, d + 1, k);
vs[k] = v;
depth[k++] = d;
```

4

```
}
}
}
// 预处理
```

5

```
void init(int V) {
// 预处理出vs、depth和id
int k = 0;
dfs(root, -1, 0, k);
// 预处理出RMQ（返回的不是最小值，而是最小值对应的下标）
```

6

```
rmq_init(depth, V * 2 - 1);
}
// 计算u和v的LCA
int lca(int u, int v) {
```

7

```
return vs[query(min(id[u], id[v]), max(id[u], id[v]) + 1)];
}
```

8

Housewife Wind （POJ No.2763）

××村里有 n 个小屋，小屋之间有双向可达的道路相连，所构成的图是一棵树。通过连接 a_i 号小屋和 b_i 号小屋的道路 i 需要花费 w_i 的时间。你一开始在 s 号小屋。请处理以下 q 个查询。

9

A：输出从当前位置移动到节点 x 所需的时间。 B：将通过道路 x 所需的时间改为 t 。

限制条件

 $1 \le n \le 100000$

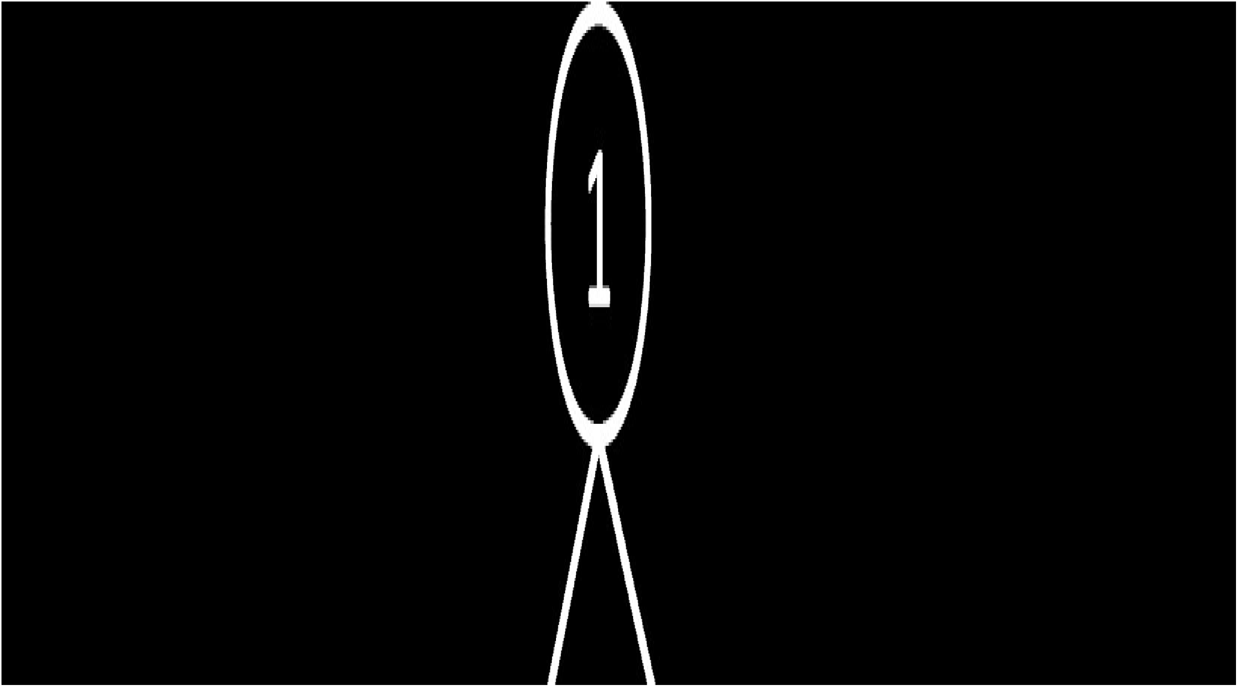
10

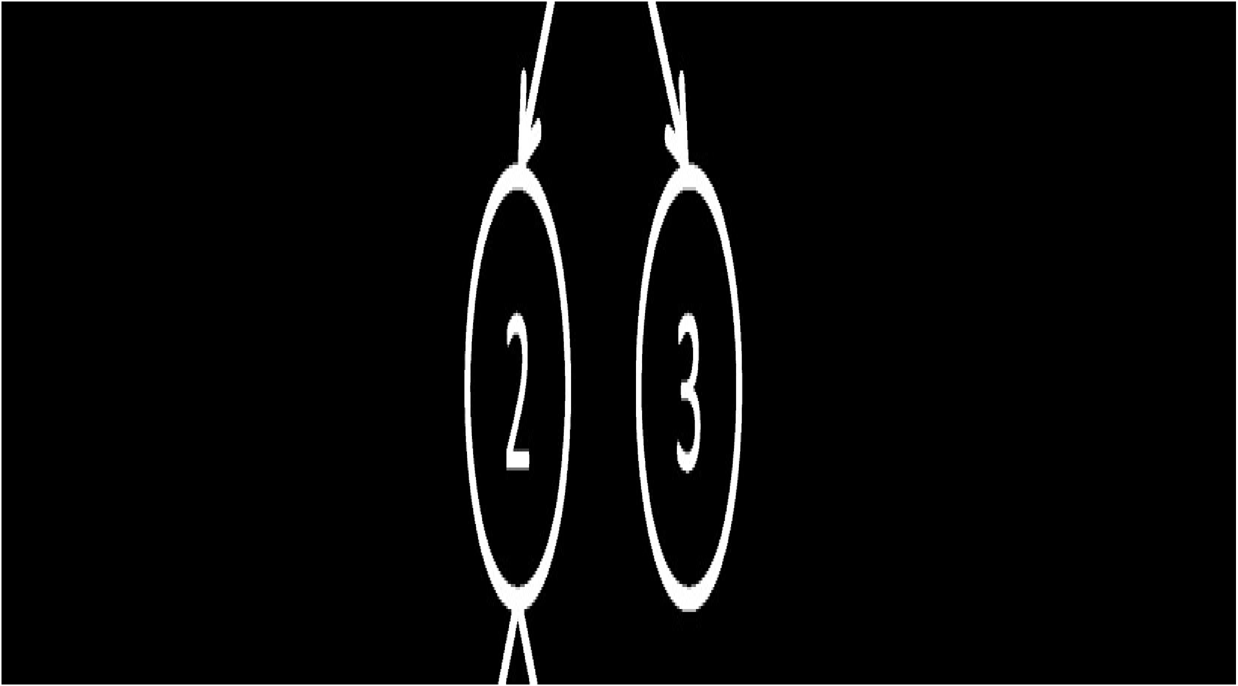
 $0 \le q \le 100000$

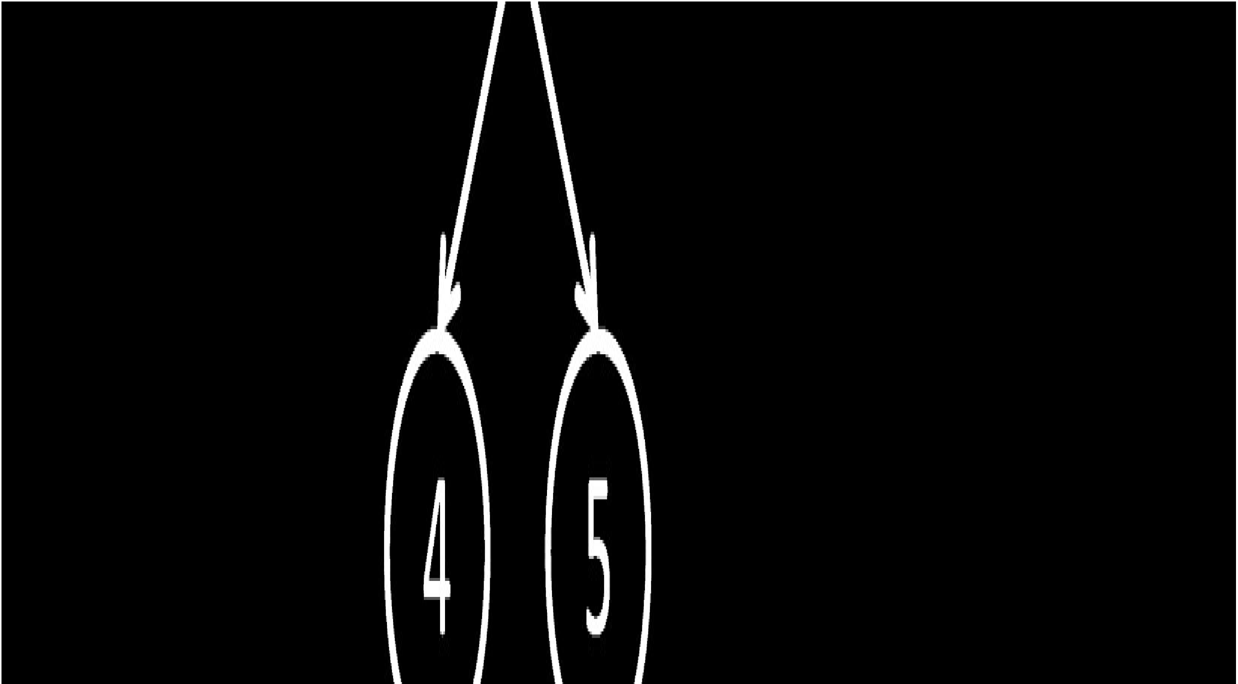
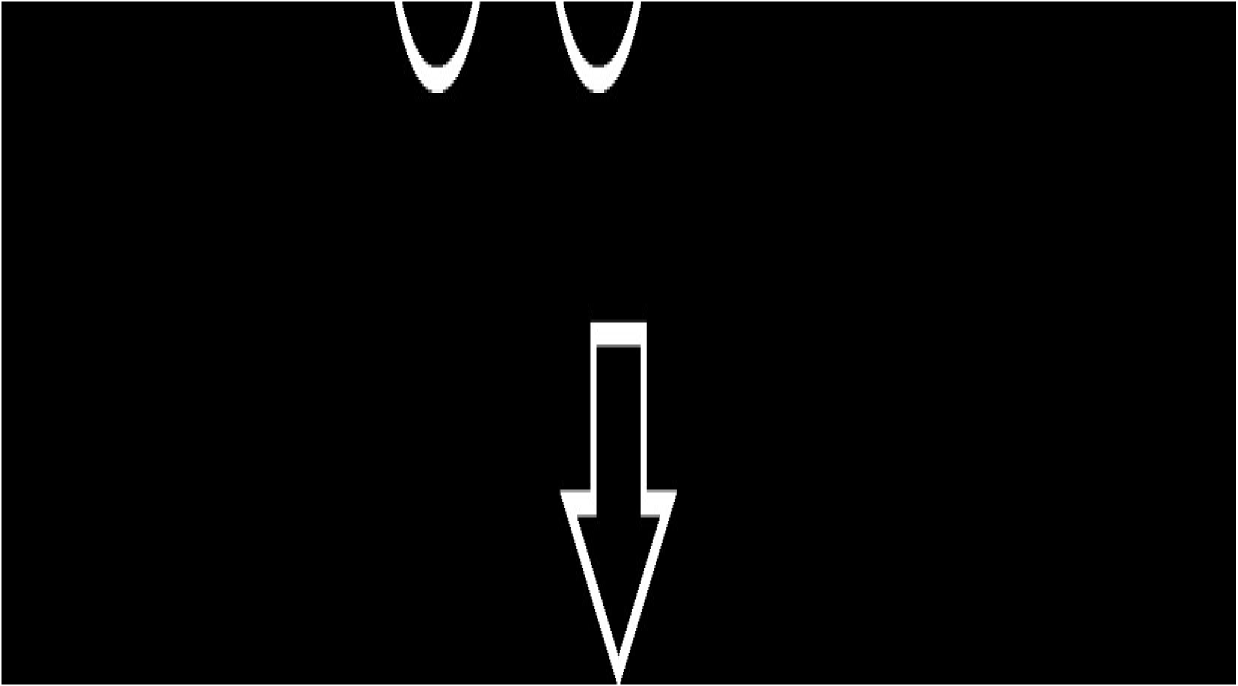
 $1 \le a_i , b_i \le n$

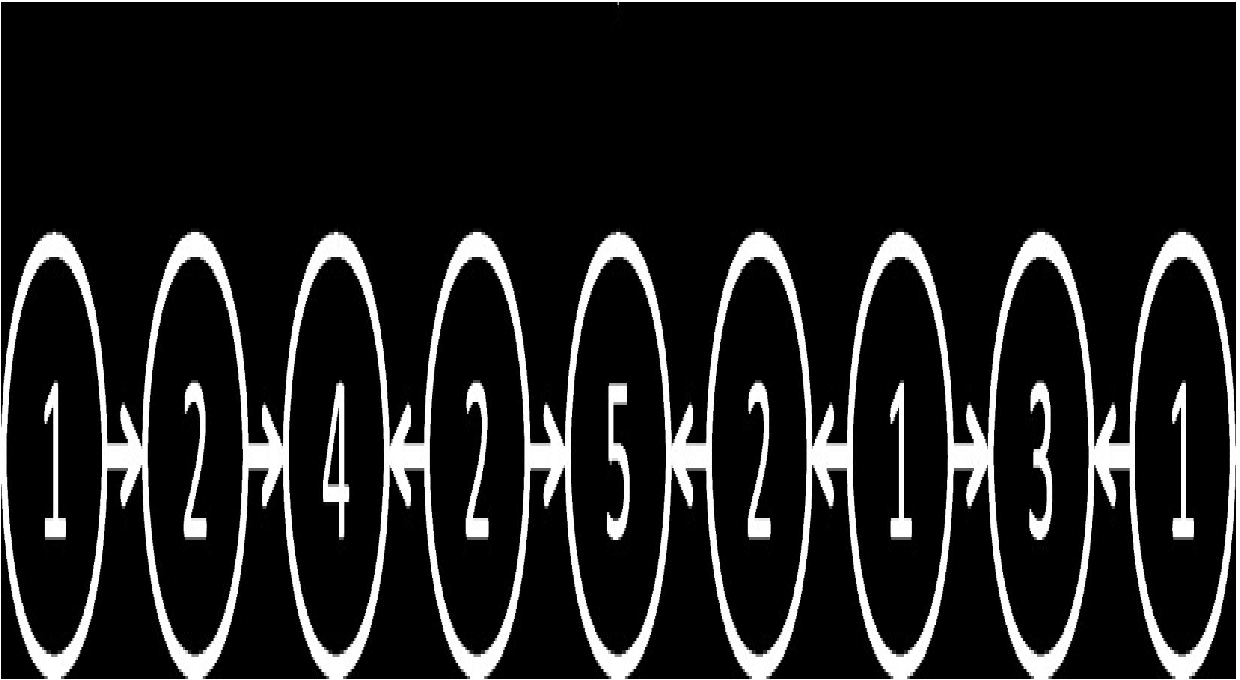
 $1 \le w_i \le 10000$

11


1


2
3


4
5




样例

输入

n = 3

q = 3

s = 1

(a, b, w) = {(1, 2, 1), (2, 3, 2)}

(查询的类型, x(, t)) = {(A, 2), (B, 2, 3), (A, 3)}

输出

从1移动到2

从2移动到3

虽然直接DFS也可以求出树上两点之间的距离，但是这对于每个A类型的查询，都要花费 $O(n)$的

时间，实在太慢了。必须利用树的特性，得到更为高效的算法。为了高效地处理A类型的查询，

可以利用二分搜索版LCA算法中介绍的技巧，记录下从每个顶点向上走2^k步的总长度。这样一来，

在 $O(\log n)$地计算LCA的同时，也可以同样 $O(\log n)$地求出到LCA的距离，因此处理A类型查询的

复杂度为 $O(\log n)$。但是，这个方法对于B类型的查询却无法高效地处理。

因此，让我们先考虑一下图是链状时这一简单的情况。假设 i 和 $i+1$两点之间的边的长度为 w_i，则

两点 u 和 v ($u < v$)之间的距离为

v 1

□

□ w i

i □ u

只要用BIT，不论是A类型的查询还是B类型的查询，都能够在 $O(\log n)$时间内处理。

在回过头来考虑树的情况。因为树中连接两点的路径是唯一的，如果我们对顶点进行合理排列的

话，能否像链状时那样，进行类似的处理呢？考虑利用RMQ计算LCA时所用的，按DFS访问的顺

序排列的顶点序列。这样， u 和 v 之间的路径，就是在序列中 u 和 v 之间的所有边减去往返重复的部

分得到的结果。

将树转为序列

于是，只要令边的权重沿叶子方向为正，沿根方向为负，那么往返重复的部分就自然抵消了，于

1

是有

(*u* , *v* 之间的距离)=(从 *LCA* (*u* , *v*)到 *u* 的边的权重和)+(从 *LCA* (*u* , *v*)到 *v* 的边的权重和) **2**

同链状的情况一样，利用BIT的话，计算权重和和更新边权都可以在 *O* (log *n*)时间内办到，而LCA 也能够在 *O* (log *n*)时间内求得。

```
struct edge { int id, to, cost; };
```

3

```
int n, q, s;

int a[MAX_V - 1], b[MAX_V - 1], w[MAX_V - 1];

int type[MAX_Q]; // 0：A类型，1：B类型

int x[MAX_Q], t[MAX_Q];
```

4

```
vector<edge> G[MAX_V]; // 图的邻接表表示

int root;

int vs[MAX_V * 2 - 1]; // DFS访问的顺序

int depth[MAX_V * 2 - 1]; // 节点的深度
```

```
5
int id[MAX_V]; // 各个顶点在vs中首次出现的下标
int es[(MAX_V - 1) * 2]; // 边的下标（i*2+(叶子方向:0,根方向:1))
void dfs(int v, int p, int d, int &k) {
id[v] = k;
6
vs[k] = v;
depth[k++] = d;
for (int i = 0; i < G[v].size(); i++) {
edge &e = G[v][i];
if (e.to != p) {
7
add(k, e.cost);
es[e.id * 2] = k;
dfs(e.to, v, d + 1, k);
vs[k] = v;
depth[k++] = d;
add(k, -e.cost);
8
es[e.id * 2 + 1] = k;
```

```
}
}
}
```

9

```
int stack_v[MAX_V + 10];
int stack_i[MAX_V + 10];
// 预处理
void init(int V) {
```

10

```
// 初始化BIT
bit_n = (V - 1) * 2;
// 预处理出vs、depth、id和es
int k = 0;
dfs(root, -1, 0, k);
```

11

```
// 预处理出RMQ（返回的不是最小值，而是最小值对应的下标）
```



```
rmq_init(depth, V * 2 - 1);
}
// 计算u和v的LCA
```

```
int lca(int u, int v) {

return vs[query(min(id[u], id[v]), max(id[u], id[v]) + 1)];

}

void solve() {

// 预处理

root = n / 2; // 不论以哪个节点为根都没有问题

for (int i = 0; i < n - 1; i++) {

G[a[i] - 1].push_back((edge){i, b[i] - 1, w[i]});

G[b[i] - 1].push_back((edge){i, a[i] - 1, w[i]});

}

init(n);

// 处理查询

int v = s - 1; // 当前位置

for (int i = 0; i < q; i++) {

if (type[i] == 0) {

// 从当前位置移动到x[i]

int u = x[i] - 1;

int p = lca(v, u);

// 利用BIT计算p到v和p到u的费用之和，即区间(id[p],id[v]]和
(id[p],id[u]]的权重和

printf("%d\n", sum(id[v]) + sum(id[u]) - sum(id[p]) * 2); v = u;
```

```
} else {
// 将通过道路x[i]的权重改为t[i]。
int k = x[i] - 1;
add(es[k * 2], t[i] - w[k]);
add(es[k * 2 + 1], w[k] - t[i]);
w[k] = t[i];
}
}
}
```